Differential Power Analysis on Light Weight Implementations of Block Ciphers

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Panasayya S.V.V.K Yalla
Bachelor of Science
Sir C.R. Reddy College of Engineering, 2007

Director: Dr. Jens-Peter Kaps, Professor
Department of Electrical and Computer Engineering

Summer Semester 2009
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to My parents Anantha Lakshmi and Ananda Ramayya Yalla, brother Satish and sister Swathi.

Acknowledgments

I would like to thank many people who helped during various stage of thesis work. First and foremost, I would like to thank my thesis advisor Dr Jens-Peter Kaps without whom my thesis would not be possible. Thank you for all your support and time.
I would also like to thank Dr Kris Gaj and Dr David Hwang for their valuable support. I would also thank Dr Alok Berry for all the help he rendered during crucial hours. I would also thank all CERG group members for giving a good time during my thesis. Last but not least I thank my friends Mahi, Sabari, Rajesh for support.

Table of Contents

List of Tables

List of Figures

Abstract

DIFFERENTIAL POWER ANALYSIS ON LIGHT WEIGHT IMPLEMENTATIONS OF BLOCK CIPHERS

Panasayya S.V.V.K Yalla, MS

George Mason University, 2009

Thesis Director: Dr. Jens-Peter Kaps

There is a growing interest in light weight implementation of cryptographic algorithms for low-resource ubiquitous computing devices such as a wireless sensor nodes (WSN) or radio frequency identification (RFID) tags. Most light weight cryptographic implementations are targeted to application specific integrated circuits (ASIC). However, ASICs have a high non-recurring engineering cost and longer time to market. Even though field programmable gate arrays (FPGA) are reconfigurable and have low non-recurring engineering cost, they consume more power than ASICs. Power consumption is a primary concern for light weight cryptographic applications. With the development of low-cost, low-power FPGAs for battery powered devices, they are becoming an interesting target for light weight cryptography (LWC). In this thesis we describe compact architectures of AES, Camellia, xTEA, HIGHT and Present are implemented on low-cost Xilinx Spartan3 FPGAs. Different optimization techniques are employed to minimize the area consumption by smart use of the Configurable Logic Block (CLB) structure in FPGAs. All the cipher implementations are light weight but with full strength security i.e. not 80-bit but 128-bit key length. Furthermore, differential power analysis (DPA) attacks are performed on these implementations to investigate their "natural", i.e. without any countermeasures resistance to this form of attack.

Chapter 1: Introduction

1.1 Light Weight Cryptography

Ubiquitous computing represent the third era of computing devices after mainframes and personal computer for first and second eras. Radio frequency identification (RFID) tags and wireless sensor nodes (WSN) are a few examples which are being used for automated electronic toll systems, identification tags for food products, pets, clothing and so on. This brings us close to the threshold of pervasive computing. The mass deployment of these device brings serious concerns for security and privacy. The traditional cryptographic algorithms may not be suitable for these device as they have limited memory and computational power along with serious power constraints. This led to development of new branch of cryptography called light weight cryptography. HIGHT [1] and Present [2] are the algorithms developed specifically for light weight cryptography. AES and Camellia though not considered light weight, are also being used on these devices.

1.2 Light Weight Cryptography for FPGAs

Until now light weight cryptography is targeted towards application specific integrated circuits (ASICs). ASICs involve high non-recurring cost and long time to market where as Field Programmable Gate Arrays (FPGAs) involve low non-recurring cost and less time to market. The only dominant factor favorable to ASICs is their lower power consumption, which is of primary concern for light weight cryptographic devices and their lower cost in large volumes. With the advent of low-cost and low-power FPGAs,we expect them to become popular for battery powered applications such as WSN. Hence they are a targeted for light weight cryptographic applications. Reconfigurability of FPGAs allows the system to be upgraded if ever the need arises which is not possible with ASICs. Further more, light weight crypto implementations lead to area saving over traditional implementations. This enables a designer to add crypto to an existing design at a minimal cost or to reduce the overall area consumption which might lead to cost saving as the design might now fit into a smaller, cheaper FPGA. We designed ight weight architecture of Present, Camellia, HIGHT for Xilinx Spartan3 FPGAs. For all the ciphers considered in this thesis are of full strength security i.e 128-bit key length, even though traditional light weight cryptography considers 80-bit key length to be secure.

1.3 Power Analysis Attacks

Cryptographic devices leak information in the form of sound, power and electromagnetic radiation while executing a cryptographic algorithm. This information can be used to reveal the secret key. Such type of attacks are called side channel attacks. Power analysis attacks analyzes the power consumption of cryptographic device while the cryptographic algorithm is executed. These attacks were discovered by Kocher et al [3] in 1998. He showed that the power analysis attack can reveal the secrets of smart card. This shattered the belief in the security of cryptographic devices. These attacks are of considerable concern as they can be mounted with existing hardware costing from a few hundred to a few thousand dollars. The amount of time required depends on the type of attack employed.

1.3.1 Simple Power Analysis Attacks

Simple power analysis (SPA) attacks involves direct interpretation of the power consumption measured during cryptographic operations. These attacks are more challenging.

1.3.2 Differential Power Analysis Attacks

Differential power attacks (DPA) are more powerful attacks than SPA. DPA exploits the data dependency of the power consumption of the cryptographic device. Unlike SPA, DPA requires a large number of power traces. The following are the steps involved for performing a DPA attack.

1. Choosing an intermediate result of the executed algorithm.
2. Measurement of power consumption for n number of encryptions or decryptions with different data blocks.
3. Developing a hypothetical power model.
4. Relating the hypothetical power model with actual measured power consumption values.

The intermediate result of the algorithm is chosen such that

- Its depends on the key.
- It changes with known input or output of the cipher.
- we ar able to predict the key by combining it with the known input or output and guess of the key.

Power Model

The most popular power models used are hamming weight (HW) and hamming distance (HD). HW is a simple model which count the number of '1's in a given set of data. HD counts the number of 0→1 and 1→0 transitions in a given set of data within a certain time interval. HD power model assumes that all 0→1 and 1→0 transitions consume equal power. The 0→0 and 1→1 transitions are considered to consume no power. In order to use the HD power model, consecutive data values are needed. If data V_1 changes to V_2 then relation between HD and HW is given by equation 1.1. In this thesis, we consider the HD power model for DPA attacks.

$$HD(V_1, V_2) = HW(V_1 \oplus V_2) \tag{1.1}$$

Pearson Correlation

To relate the hypothetical power model values with the measured values, correlation coefficients are used. We use the Pearson correlation coefficient which is given by equation 1.2.

$$r(P,G) = \frac{n\sum_i^n P_i G_i + - \sum_i^n P_i \sum_i^n G_i}{\sqrt{n\sum_i^n P_i^2 - (\sum_i^n P_i)^2}\sqrt{n\sum_i^n G_i^2 - (\sum_i^n G_i)^2}} \tag{1.2}$$

1.4 Thesis Organization

Chapter 2 describes common the optimization techniques employed for developing the light weight architecture of several block ciphers. Chapters 3, 4, and 5 describes how these optimization are applied to the block ciphers HIGHT, Camellia and Present block ciphers. AES[4] and xTEA[5] architecture are discussed briefly with respect to DPA attacks. The Implementation results are analyzed in Chapter 6. DPA attacks and the conclusions drawn from them are described in Chapter 8. Finally Chapter 9 describes future work.

Chapter 2: Optimization Techniques

Designing a compact architectures in FPGAs depends on effective use of architectural features provided in the targeted FPGAs. Xilinx Spatan3 family FPGAs have features like Look Up Table (LUT) based 16-bit shift register (SRL16) and distributed random access memory (DRAM) which can be employed to improve the performance and decrease the area of a given design by an order of magnitude. Light weight architectures of Camellia, Present and HIGHT are developed making use of these architectural features.

2.1 Spartan-3 Generation FPGA Architecture

2.1.1 Configurable Logic Block Structure

Spartan-3 generation consist of five fundamental programmable function elements

1. Configurable logic blocks (CLBs)
2. Input/output blocks (IOBs)
3. Block RAMs
4. Multiplier blocks
5. Digital clock manager (DCM) blocks

The main logic resource for implementing synchronous as well as combinational circuits are constituted by CLBs. Each CLB consists of four interconnected slices which are grouped in pairs and organized as columns as shown in Fig 2.1. The pair on left side of the CLB supports both logic and memory functions and is called SLICEM. The other pair on the right side of the CLB supports only logic and is called SLICEL. The structure of SLICEL

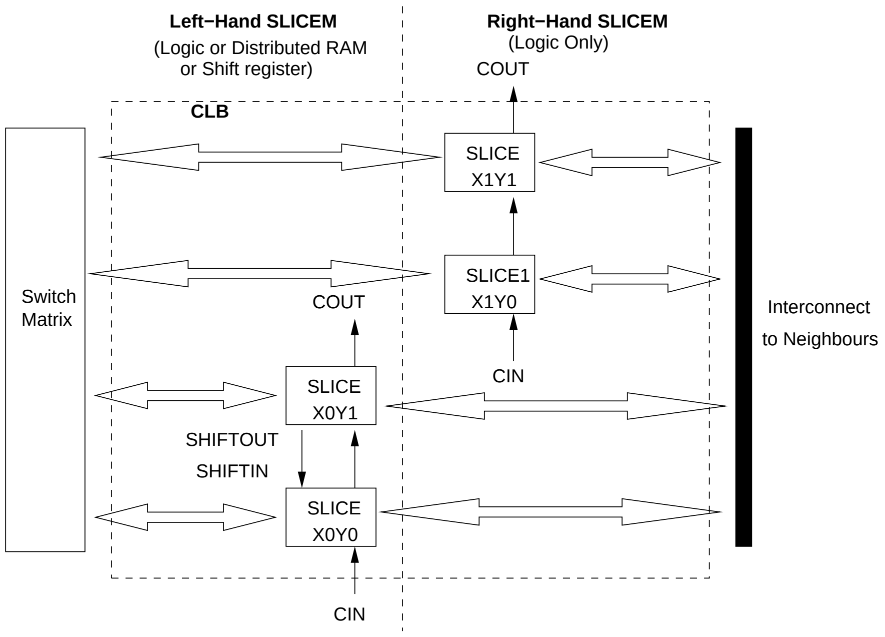

Figure 2.1: Arrangement of Slices within the CLB

and SLICEM are shown in Fig 2.2 and Fig 2.3 respectively. Both SLICEL and SLICEM have the following features which provide logic, arithmetic and ROM functions.

- Two 4-input LUT function generators, F and G
- Two storage elements (Flipflops)
- Two wide-function Multiplexers
- Carry and arithmetic logic

Apart from this SLICEM LUTs provide additional features namely 16x1 distributed RAM blocks called RAM16 and 16-bit shift register called SRL16.

Shift Register (SRL16)

The SLICEM LUT can be configured as a 16-bit shift register called SRL16 without using the flipflop available in each slice. Fig 2.4 shows the basic cell structure of SRL16. The F-LUT and G-LUT in the SLICEM shown in Fig 2.3 are the basic components of SRL16. Within SLICEM, LUTs cascade from the G-LUT MC15 output to the F-LUT DI input

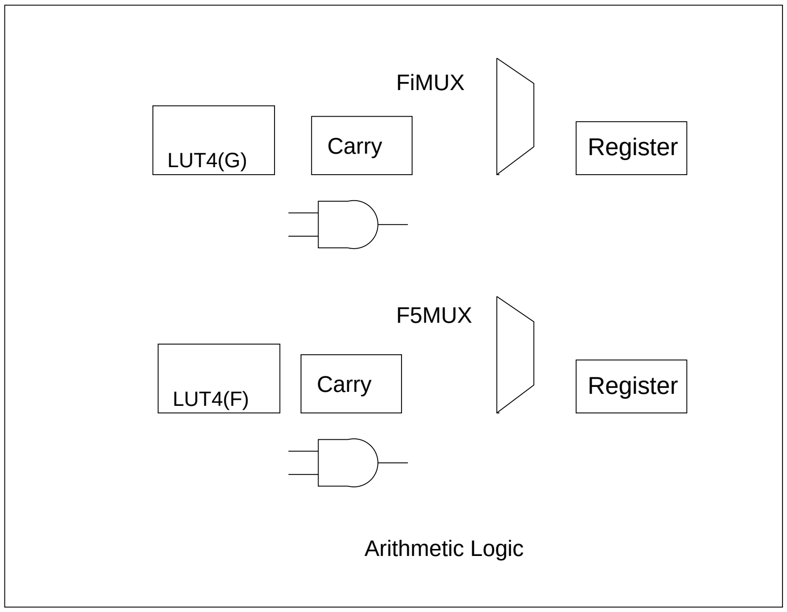

Figure 2.2: SLICEL

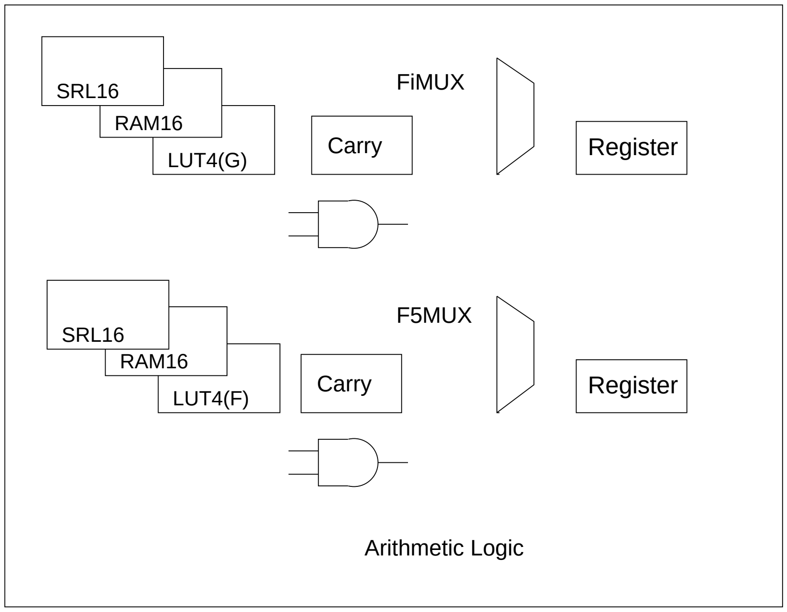

Figure 2.3: SLICEM

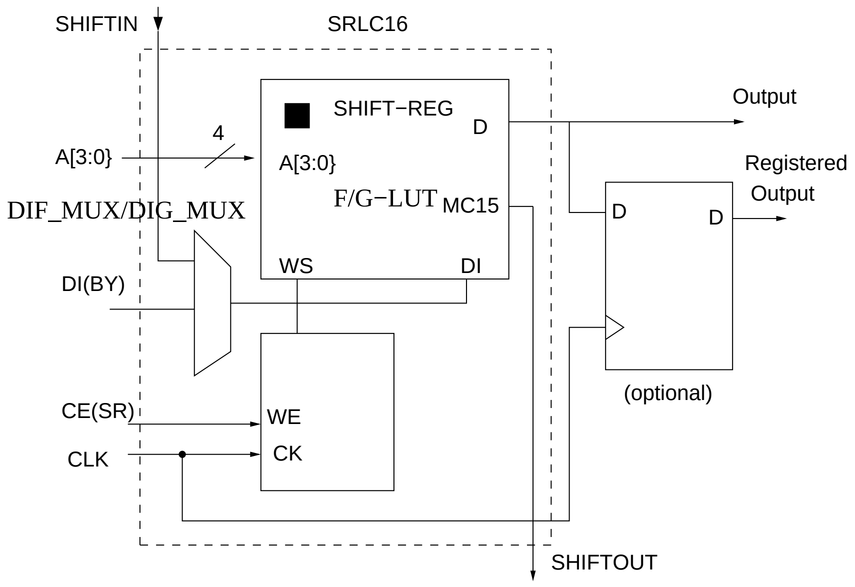

Figure 2.4: Logic Cell SRL Structure

through $DIFMUX$. The SHIFTIN and SHIFTOUT lines allow cascading SLICEMs using $DIGMUX$ to form larger shift registers. Each shift register provides a shift output MC15 for the last bit in each LUT in addition to addressable access to any bit in the shift register through normal D output. The shift register input can come from a dedicated SHIFTIN signal and Q15/MC15. The addressable D output is available in all SRL primitives while Q15/MC15 signal that can drive SHIFTOUT are only available in SRLC16 primitive. SRLC16 are cascade-able shift register C stands for cascade-able. Each CLB resource can be configured as a 64-bit shift register using four LUTs. The synthesis tool infers SRL16s when a shift register is described in hardware descriptive language (HDL). If a reset is placed, synthesis tool infers them as flipflops. The number of slices required for implementing a shift register depends on number of taps required for reading or writing as additional taps require additional flipflops. Taps are positions of shift register where the data can be written into or read out. These taps are configured as flipflops.

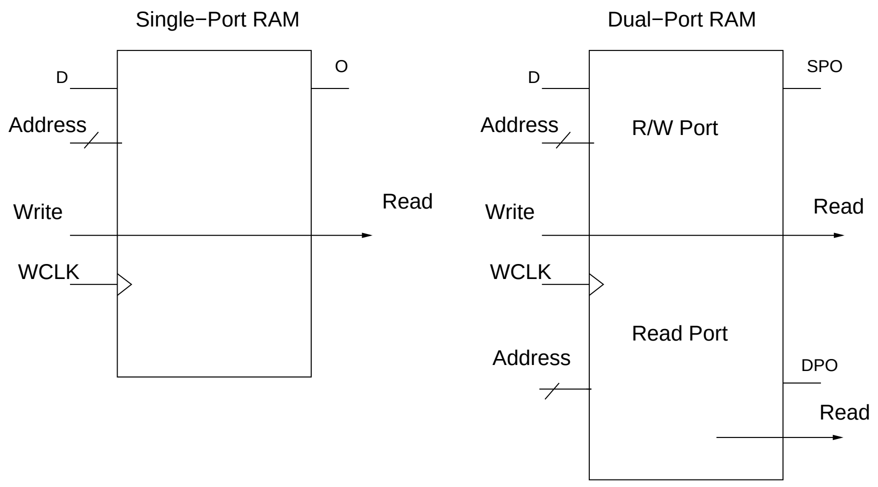

Figure 2.5: Single-Port and Dual-Port Distributed RAM

Distribute RAM (RAM16)

The SLICEM LUT can be configured as 16x1-bit synchronous RAMs called as distributed RAM (DRAM) or RAM16. These LUTs are cascade-able for realizing deeper memories with minimal penalty on timing. Distribute RAM support two types of memory

- Single-port RAM with synchronous write and asynchronous read. Synchronous reads are configured using the flipflop associated with it.

- Dual-port RAM with one synchronous write and two asynchronous read ports. Synchronous reads are possible. The second read port is an independent read port.

Fig 2.5 shows the single- and dual-port RAMs. The flipflop in SLICEM allows the capturing of output from distributed RAM. Each CLB can be configured as a 64-bit single port RAM or 32-bit dual-port RAM. Use of RAM16s allows to store large amount of data in small RAM blocks. DRAMs offer fast and localized memory. They can be inferred or instantiated directly in design depending on the specific logic synthesis tool used. Most of the tools infer them based on the hardware description of RAMs.

2.2 Optimizations

Area efficient architectures are developed using of the features described in 2.1.1. Choosing the appropriate feature for implementing a specific component in the design results in an efficient and compact architecture. The most area consuming are data and key storage components. DRAM and shift register are two ways of efficient memory implementation. Use of shift registers for storing of data does not involve addressing which makes the control logic simpler. However, accessing the intermediate data values other than at the taps of the shift register is not possible. In that case the best suitable one is DRAM. Shifting of data in DRAM is complicated. The best suited one is shift register. DRAM involve addressing which increases the complexity of the control logic.

2.2.1 Plaintext and key Storage

For developing light weight architectures, the algorithm implementations are scaled down to either 8-bit or 16-bit implementations. Key and data are loaded either 8-bits or 16-bits depending on the implementation. Loading into shift register is simpler as the number of control bits needed are less as compared to DRAM which needs addressing. The size of the address increases with increase in number of words to be stored in DRAM, where as control bits of shift register are independent of size of the data it stores. However, some cipher require intermediate data values. For example in the case of Camellia, the different bytes of the data are required to compute a single byte. This makes use of DRAM more appropriate. In other two cipher implementations Present and HIGHT, plaintext is stored in shift registers as data is needed in a regular order. The key scheduling in ciphers Camellia, HIGHT and Present involve shifting of key which make use of shift register more apt. However in HIGHT, key is need in a different order during initial and final transformations compared to round operation which is difficult using shift register. In this case DRAM is used to store the key.

2.2.2 Finite State Machine

Finite State Machines (FSM) are used for realizing the control logic of complex systems. Traditionally, FSM are implemented using flipflops and programmable logic. However this type of FSM implementation is complex and is not efficient. Use of RAM blocks for sequential logic led to ROM-based FSM implemented which proved to be efficient [6], [7], [8]. The control signals for each operation are clubbed as one big control word. These control words are stored in a memory location which can be accessed by an address. DROM are inferred by holding the write signal low for DRAMs. ROM- based FSM have additional advantages. The maximum frequency at which a ROM-based FSM operates is independent of complexity of the circuit. This method is also proved be power saving [9]. For HIGHT and Present architectures, the control signals are generated by using counter and some additional logic.

2.2.3 Other components

The size of control word is reduced by removing any control signals which are repeated many times. Control signals for round operation are repeated sequence of operations. This control signal are remove from the main control word reducing the size ROM needed for implementing the FSM. This is applied for camellia where the control signals for round function f are generated by another sub-controller called F-controller.

Chapter 3: HIGHT

3.1 Introduction

HIGHT [1] a is block cipher developed by a group from Korea University, National Security Research Institute (NSRI) and Korea Information Security Agency (KISA) in 2006. HIGHT (HIGH security and light weighT) is a 64-bit block cipher with 128-bit key length and 32-rounds. The round function consists of a generalized Feistel structure with simple operations such as XOR, addition modular 2^8, and left bitwise rotation. The absence of traditional substitution layer,its feistel structure and byte oriented operations make it suitable for low-cost, low-power and ultra-light implementations. The original design presented in [1] was implemented on ASICs with 3048 gates. The HIGHT algorithm was modified [10] to reduce the critical path in the key scheduler which also reduced the area to 2608 gates. Initial security analysis [1] proved 19 rounds of HIGHT to be secure. Further analysis [11] proved 28 rounds of hight to be secure.

3.2 Algorithm

3.2.1 Notations

The following notations are used in describing HIGHT. The 64-bit plain text P is split into eight 8-bit blocks $P_7, \cdots, P_0$. The intermediate values X and cipher text C are also split into eight 8-bit blocks $X_7, \cdots, X_0$ and $C_7, \cdots, C_0$ respectively. The 128-bit master key MK is a concatenation of sixteen 8-bit blocks. The whitening keys and subkeys are denoted by WK and SK each is of 8-bit size.

$$\begin{aligned} P &= P_7 \parallel P_6 \parallel ......P_1 \parallel P_0 \\ C &= C_7 \parallel C_6 \parallel ......C_1 \parallel C_0 \\ X &= X_7 \parallel X_6 \parallel ......X_1 \parallel X_0 \\ MK &= MK_{15} \parallel MK_{14} \parallel ......MK_1 \parallel MK_0 \end{aligned} \tag{3.1}$$

The mathematical operations used in HIGHT are denoted as follows

$\boxplus$	addition mod2^8
$\boxminus$	subtraction mod2^8
$\oplus$	XOR(eXclusive OR)
$A^{\lll s}$	s-bit left rotation of a 8-bit value of A

The HIGHT encryption consist of key schedule, initial transformation, round function and final transformation. The algorithm for encryption process can be described as follows

```
HightEncryption(P,MK) {
KeySchedule(MK,WK,SK);
HightEncryption(P,WK,SK){
InitialTransformation(P,$X_0$,$WK_3$,$WK_2$,$WK_1$,$WK_0$);
**for** $i = 0$ to 31 **do**
  RoundFunction ($X_i$,$X_{i+1}$,$SK_{4i+3}$,$SK_{4i+2}$,$SK_{4i+1}$,$SK_{4i}$);
**end for**}
FinalTransformation($X_{32}$,C,$WK_7$,$WK_6$,$WK_5$,$WK_4$); } }
```

3.2.2 Key schedule

Key scheduling of HIGHT involves two algorithms WhiteningkeyGeneration and Subkey-Generation.The key schedule algorithm is as follows

```
  KeySchedule(MK,WK,SK){
WhiteningkeyGeneration(MK,WK);
SubkeyGeneration(MK,SK);}
```

Whiteningkey Generation

WhiteningkeyGeneration generates 8 whitening key bytes WK$_0$,WK$_1$,$\cdots$,WK$_7$ for initial and final transformations. Whitening keys are used to avoid direct revealing of inputs to F$_0$ and F$_1$ during first and last rounds. The whitening key algorithm is as follows

```
WhiteningkeyGeneration{
for $i = 0$ to 7 do
  if $0 \leq i \leq 3$ then
    $WK_i \leftarrow MK_{i+12}$;
  else
    $WK_i \leftarrow MK_{i-4}$;
  end if
end for}
```

Subkey generation

SubkeyGeneration generates 128 subkey bytes SK$_0$,SK$_1$,$\cdots$,SK$_{127}$ for 32-round functions. The algorithm SubkeyGeneration uses ConstantGeneration which generates 128 7-bit constants $\delta_0, \delta_1, \cdots, \delta_{127}$, for generating the subkeys SK$_0$,SK$_1$,$\cdots$,SK$_{127}$. δ_0 has a fixed value of 1011010_2 which is the initial state (s_0,$\cdots$,s_6) of the 7-bit LFSR h. The characteristic polynomial of h is x^7+x^3+1 in $\mathbb{Z}_2$ with a period of 2^7-1=127. Using a LFSR h for subkey generation enhances the randomness of subkey bytes. This provides improved resistance against slide attack. The ConstantGeneration algorithm uses the LFSR h to produce 128 δ values as follows

```
ConstantGeneration{
$S_0 \leftarrow 0$; $S_1 \leftarrow 1$; $S_2 \leftarrow 0$; $S_3 \leftarrow 1$;
$S_4 \leftarrow 1$; $S_5 \leftarrow 0$; $S_6 \leftarrow 1$;
$\delta_0 \leftarrow S_6 \| S_5 \| S_4 \| S_3 \| S_2 \| S_1 \| S_0 \|$;
for $i = 1$ to 127 do
  $S_{i+6} \leftarrow S_{i+2} \oplus S_{i-1}$;
  $\delta_i \leftarrow S_6 \| S_5 \| S_4 \| S_3 \| S_2 \| S_1 \| S_0 \|$;
end for}
```

The algorithm for Subkey generation is as follows

```
SubkeyGeneration(MK,SK){
Run ConstantGeneration
for $i = 0$ to 7 do
  for $j = 0$ to 7 do
    $S_{16i+j} \leftarrow MK_{(j-i) \bmod 8} \boxplus \delta_{16i+i}$;
  end for
  for $j = 0$ to 7 do
    $S_{16i+j+8} \leftarrow MK_{((j-i) \bmod 8)+8} \boxplus \delta_{16i+j+8}$;
  end for
end for}
```

3.2.3 Initial Transformation

InitialTransformation uses the four whitening key bytes WK_0, WK_1, WK_2 and WK_3 to transform a plain text P into the input of first RoundFunction, $X_0 = X_{0,7} \| X_{0,6} \|......X_{0,1} \|$

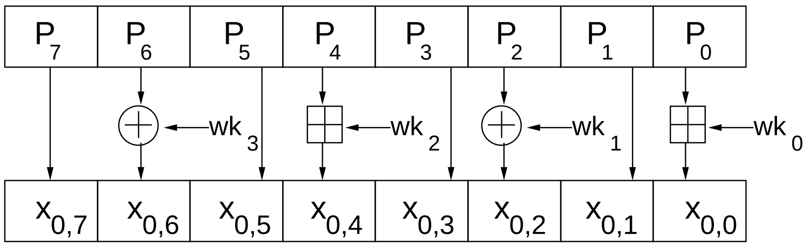

Figure 3.1: Initial Transformation

$X_{0,0}$. The InitialTransformation performs a XOR or modular addition on the input which is shown in Fig 3.1. as follows

```
InitialTransformation(P,$X_0$,$WK_3$,$WK_2$,$WK_1$,$WK_0$) {
$X_{0,0} \leftarrow P_0 \boxplus WK_0$; $X_{0,1} \leftarrow P_1$; $X_{0,2} \leftarrow P_2 \oplus WK_1$; $X_{0,3} \leftarrow P_3$;
$X_{0,4} \leftarrow P_4 \boxplus WK_2$; $X_{0,1} \leftarrow P_5$; $X_{0,6} \leftarrow P_6 \oplus WK_1$; $X_{0,3} \leftarrow P_3$; }
```

3.2.4 Round Function

RoundFunction uses two auxiliary function F_0 and F_1 along with addition mod 8 and XOR operations. The alternative combination of addition mod8 and XOR is used in the round function. This makes it more resistant to existing attacks. The two functions F_0 and F_1, described in equations 3.2 and 3.3 respectively, provide bitwise diffusion which is similar to linear transformation from $GF(2)^8$ to $GF(2)^8$. The RoundFunction transforms the input X_i=$X_{i,7} \parallel X_{i,6} \parallel \cdots X_{i,1} \parallel X_{i,0}$ into X_{i+1}=$X_{i+1,7} \parallel X_{i+1,6} \parallel \cdots X_{i+1,1} \parallel X_{i+1,0}$ for $i=$ 0,1$\cdots$,30,31 which is shown in Fig 3.2

$$F_0(x) = x^{\lll 1} \oplus x^{\lll 2} \oplus x^{\lll 7} \tag{3.2}$$

$$F_1(x) = x^{\lll 3} \oplus x^{\lll 4} \oplus x^{\lll 6} \tag{3.3}$$

```
RoundFunction($X_i$,$X_{i+1}$,$SK_{4i+3}$,$SK_{4i+2}$,$SK_{4i+1}$,$SK_{4i}$) {
```

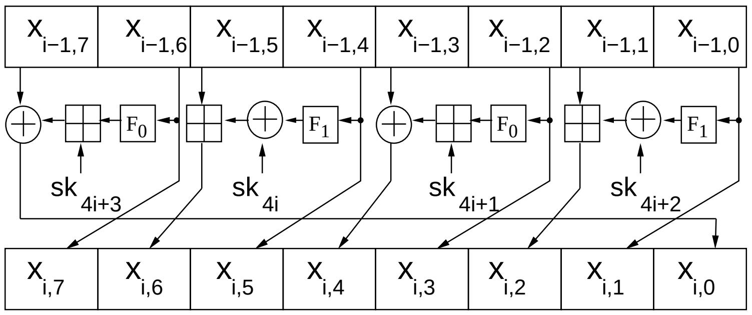

Figure 3.2: Round Function

$X_{i+1,1} \leftarrow X_{i,0}$; $X_{i+1,3} \leftarrow X_{i,2}$;

$X_{i+1,5} \leftarrow X_{i,4}$; $X_{i+1,7} \leftarrow X_{i,6}$;

$X_{i+1,0}= X_{i,7} \oplus (F_0(X_{i,6}) \boxplus SK_{4i+3})$;

$X_{i+1,2}= X_{i,1} \boxplus (F_1(X_{i,0}) \oplus SK_{4i+2})$;

$X_{i+1,4}= X_{i,3} \oplus (F_0(X_{i,2}) \boxplus SK_{4i+1})$;

$X_{i+1,6}= X_{i,5} \boxplus (F_1(X_{i,4}) \oplus SK_{4i})$; }

3.2.5 Final Transformation

In the Final transformation, the data is shifted towards right and transforms $X_{32}=X_{32,7} \parallel X_{32,6} \parallel \cdots X_{32,1} \parallel X_{32,0}$ into cipher text C by using the four whitening key bytes WK_4, WK_5, WK_6 and WK_7 as shown in Fig 3.3

$(X_{32},C,WK_7,WK_6,WK_5,WK_4)$ {

$C_0 \leftarrow X_{32,1} \boxplus WK_4$; $C_1 \leftarrow X_{32,2}$; $C_2 \leftarrow X_{32,3} \oplus WK_5$; $C_3 \leftarrow X_{32,4}$;

$C_4 \leftarrow X_{32,5} \boxplus WK_6$; $C_5 \leftarrow X_{32,6}$; $C_6 \leftarrow X_{32,7} \oplus WK_7$; $C_7 \leftarrow X_{32,0}$;

3.3 Light weight Architecture of HIGHT

Our light weight architecture of HIGHT is achieved by applying the optimizations described in Chapter 2. Implementation of different components used in this architecture were described.

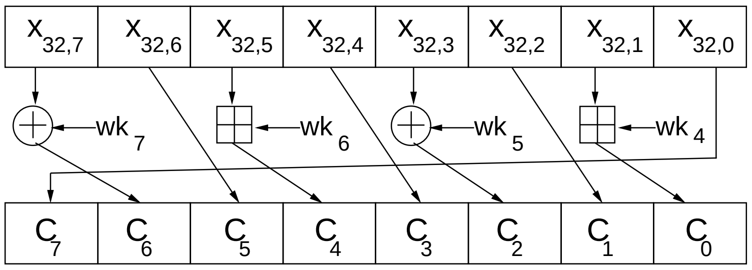

Figure 3.3: Final Transformation

3.3.1 Data storage

RoundFunction involves shifting of data for which shift register are bested suited one. Shift registers have an advantage over DRAMs as they do not need addressing. This makes the control logic simple. The shift register used for storing the data is a 64-bit shift register which performs an 8-bit left-circular shift. In order to generate one byte, RoundFunction needs either one byte or two bytes of consecutive data. The shift register used in this architecture has 8-bit datain and 16-bit dataout. The first subkey generated for i^{th} round operation is used in computing $X_{i+1,6}$ from $X_{i,5}$ and $X_{i,4}$. So taps are placed at 5^{th} and 4^{th} bytes. The round function computes $X_{i+1,6}$ using $X_{i,5}$ and $X_{i,4}$ and writes the result into $X_{i,5}$ as it is no longer required for computation of other bytes. This saves extra memory needed for temporary storage. The next byte $X_{i+1,5}$ is $X_{i,4}$ which can be achieved by shifting the data by one byte as $X_{i,4}$ is next to $X_{i+1,6}$. Now, the 5^{th} and 4^{th} byte locations contains $X_{i,3}$ and $X_{i,2}$ which are required for computing $X_{i+1,4}$. The same process is repeated to compute all bytes. It takes 8-clock cycles to compute all bytes for one RoundFunction but the data is misaligned by two bytes. The data is aligned by shifting the data by 2-bytes which involves 2-clock cycles. With this, the total clock cycles required for one RoundFunction are 10.

Pre- and Post-whitening Rounds

The initial and the final transformations are performed by using the datapath of round function with use of two extra multiplexers. The initial transformation is performed while the data is being loaded into the shift register which save clock cycles. When all the data is loaded, data is misaligned by 2-bytes as the input tap to shift register is at 6^{th} byte position. The data is shifted by 2-byte to aligned it. In order to accomplish final transformation, the data should be shifted by 5-bytes requiring 5 extra clock cycles.

3.3.2 Key Storage and Scheduling

The 128-bit key is stored in a Single-Port RAM. Using two 64 bit shift registers for storing the key would make the control logic more simpler. But generation of whitening keys would involves more clock cycles than using a RAM. The subkeys and whitening can be generated by using two 3-bit counter and a 2x1 multiplexer. The selection bit and most significant bit of the key address are generated by using a 4-bit shift register.

3.3.3 Control Logic

Control logic for this architecture is much simpler compared to Present and Camellia described in Chapters 5 and 4 respectively as shift register need few control signals. The control signals for all operation are generated by using 4-bit and 6-bit counters. The 4-bit counter is used for counting the intermediate states of round operation. So the only extra hardware needed for the control logic is 6-bit counter along with some logic functions. The top level block diagram is shown in Fig 3.4.

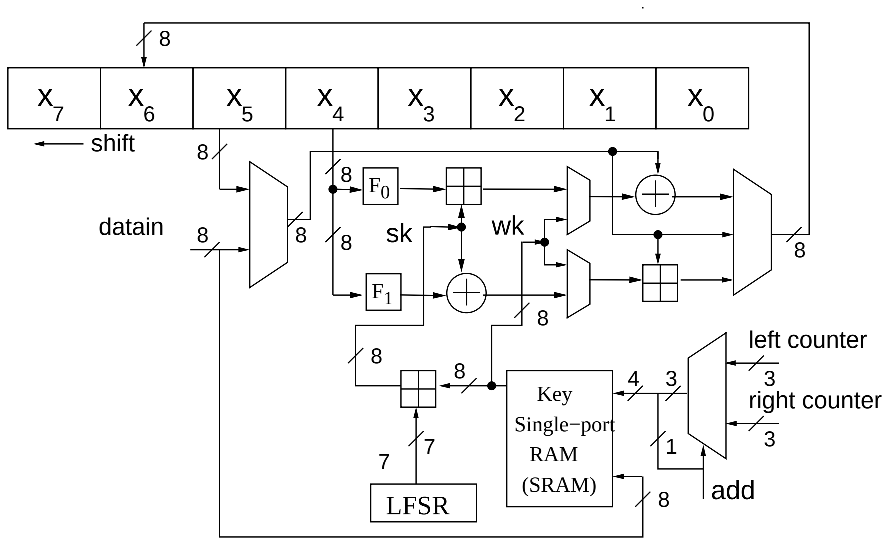

Figure 3.4: Top level block diagram of HIGHT

Chapter 4: Camellia

4.1 Introduction

The Camellia algorithm [12] was jointly developed by Nippon Telegraph and Telephone Corporation (NTT) and Mitsubishi in 2000. It was designed for a wide range of design platforms from low power and limited resources to high performance on multiple platforms. However, the main design goal was security. The New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project has nominated Camellia as a strong block cipher in 2003 [13]. The structure of Camellia provides features for a compact design. Several different attacks were performed successfully only on reduced round versions of Camellia. An impossible differential cryptanalysis on reduced round Camellia is described in [14], collision attacks in [15, 16]. The best know attack can break 9-rounds of Camellia with 128-bit key [15].

4.2 Camellia

Camellia [12] is a 128-bit block cipher which supports key lengths of 128, 192 or 256 bits. In this paper, we describe an implementation with 128-bit key length. The Camellia algorithm uses a Feistel network with pre-whitening before first and post-whitening after last rounds. The functions, FL and FL^{-1} are inserted after 6th and 12th round introduce non-regularity. The block diagram of 128-bit encryption can be seen in Fig 4.1. The F-function contains a Substitution-Permutation Network (SPN) which is composed of non-linear S-function and linear P-function. The S-function consists of 8 S-Boxes which are selected from four different types. The P-function is comprised of byte permutations. The block diagram of the F-function is shown in Fig 4.2. The key schedule generates round keys of 64-bit size by shifting the original key K_L and the modified key K_A. Computation of K_A is described in the Section 4.2.3.

4.2.1 Notations

X_L	left-half data of X.
X_R	right-half data of X.
$\oplus$	bitwise exclusive-OR operation.
$\|$	concatenation of two operands.
$\lll_n$	circular rotation to left by n bits
$\ggg_n$	circular rotation to right by n bits
$\cup$	bitwise AND operation
$\cap$	bitwise OR operation

4.2.2 Encryption for 128-bit key

The 128-bit plain text M_{128} is XORed with pre-whitening key $kw_1 \| kw_2$ and separated into two halves L_0 and R_0 each of 64-bit size. L_0 is then passes through the F-function where it is XORed with round key k_0 The result is applied to the S-Boxes and output of S-box to P-function which is XORed with right half of the data R_0. At the end of each routine the left half and the right half of the data are swapped. The same process is repeated for all the 18 rounds. The 6^{th} and 12^{th} rounds, the left half of the data $L_r^{'}$ is given to FL and right half $R_r^{'}$ to FL^{-1}. The round functions can be described as follows
For r=1 to 18 except r=6 and 12

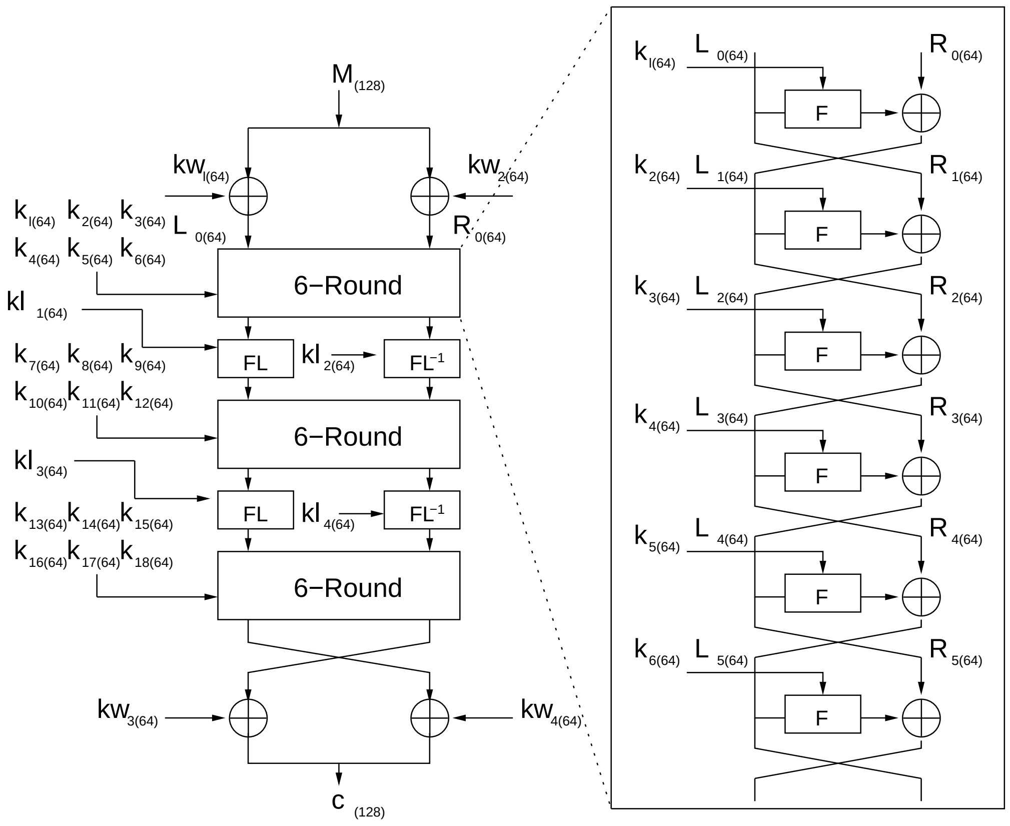

Figure 4.1: Block diagram of 128-bit key encryption

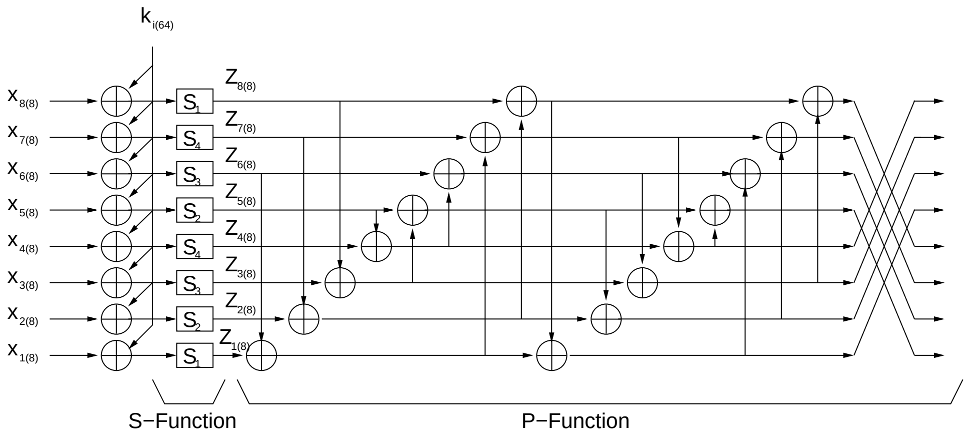

Figure 4.2: F-function

$$
\begin{aligned}
L_r &= F(L_{r-1}, k_r) \oplus R_{r-1} \\
R_r &= L_{r-1}
\end{aligned}
$$

For r=6 and 12

$$
\begin{aligned}
L_r^{'} &= F(L_{r-1}, k_r) \oplus R_{r-1} \\
R_r^{'} &= L_{r-1} \\
L_r &= FL(L_r, kl_{2r/6-1}) \\
R_r &= FL^{-1}(R_r^{'}, kl_{2r/6}) \\
C &= (R_{18} \parallel L_{18}) \oplus (kw_3 \parallel kw_4)
\end{aligned}
$$

4.2.3 Key schedule

In the first phase of the key schedule, the modified key K_A is computed from the original key K_L. In the second phase, round keys are generated through rotation of K_L or K_A by 15 or 17-bits according to Table 4.2. K_A is computed by passing K_L through 4 rounds of the same Feistel network which is used for encryption with XOR of K_L after the 2^{nd} round. The round keys used are four constant shown in Table 4.1.

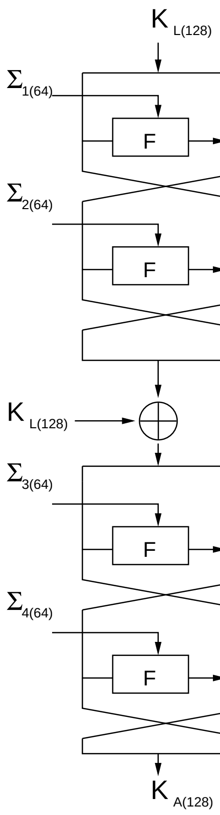

Figure 4.3: Key scheduling

Table 4.1: Key schedule constants

$\sum_1$	0xA09E667F3BCC908B
$\sum_2$	0xB67AE8584CAA73B2
$\sum_3$	0xC6EF372FE94F82BE
$\sum_4$	0x54FF53A5F1D36F1C

Table 4.2: Round keys for 128-bit keys

Round	round key	value
Pre-whitening	kw_1	$(K_L \ggg_0)_L$
Pre-whitening	kw_2	$(K_L \ggg_0)_R$
F(Round 1)	k_1	$(K_A \ggg_0)_L$
F(Round 2)	k_2	$(K_A \ggg_0)_R$
F(Round 3)	k_3	$(K_L \ggg_{15})_L$
F(Round 4)	k_4	$(K_L \ggg_{15})_R$
F(Round 5)	k_5	$(K_A \ggg_{15})_L$
F(Round 6)	k_6	$(K_A \ggg_{15})_R$
FL	kl_1	$(K_A \ggg_{30})_L$
FL^{-1}	kl_2	$(K_A \ggg_{30})_R$
F(Round 7)	k_1	$(K_L \ggg_{45})_L$
F(Round 8)	k_2	$(K_L \ggg_{45})_R$
F(Round 9)	k_3	$(K_A \ggg_{45})_L$
F(Round 10)	k_4	$(K_L \ggg_{60})_R$
F(Round 11)	k_5	$(K_A \ggg_{60})_L$
F(Round 12)	k_6	$(K_A \ggg_{60})_R$
FL	kl_1	$(K_L \ggg_{77})_L$
FL^{-1}	kl_2	$(K_L \ggg_{77})_R$
F(Round 13)	k_1	$(K_L \ggg_{94})_L$
F(Round 14)	k_2	$(K_L \ggg_{94})_R$
l F(Round 15)	k_3	$(K_A \ggg_{94})_L$
F(Round 16)	k_4	$(K_A \ggg_{94})_R$
F(Round 17)	k_5	$(K_L \ggg_{111})_L$
F(Round 18)	k_6	$(K_L \ggg_{111})_R$
Post-whitening	kw_3	$(K_A \ggg_{111})_L$
Post-whitening	kw_4	$(K_A \ggg_{111})_R$

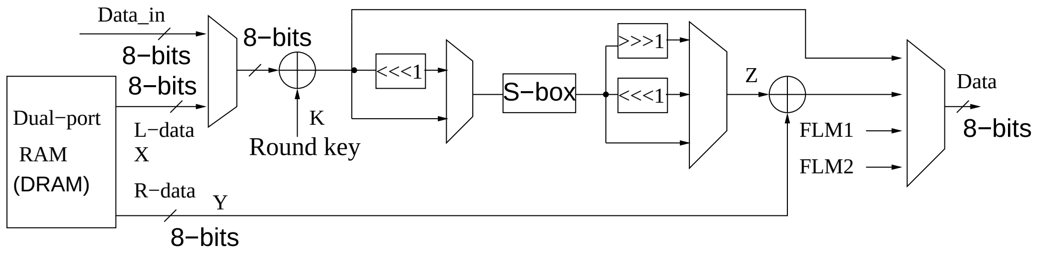

Figure 4.4: Top level Block Diagram

4.3 Compact Architecture of Camellia

Our goal is to design a very compact architecture for small area with an acceptable throughput. We choose to implement our architecture on Xilinx spartan-3 family FPGA devices. Our architecture uses a 8-bit datapath and does both encryption and key scheduling. Fig 4.4 shows the top level block diagram of our architecture. We tried different implementation strategies for several component used in the architecture to get the best results.

4.3.1 S-Boxes and F-Function

The S-Boxes S_2, S_3, S_4 can be derived from S-Box using the equation through equations 4.1, 4.2 and 4.3. This can be realized in hardware through one S-Box and two multiplexers. Hence instead of 8 S-Boxes, only one S-box is required which reduces the area by 85% .

$$S_2(x) = S_1(x) <<<_1 \tag{4.1}$$

$$S_3(x) = S_1(x) >>>_1 \tag{4.2}$$

$$S_4(x) = S_1(x <<<_1) \tag{4.3}$$

In this architecture, a dual port 16x8 Distributed RAM (DRAM) is used for storing the data which reduces the area by approximately 75% compared to using a 128-bit register.

In this implementation, the 64-bit F-function is broken down into several of 8-bit operations. The XOR of 8-bits of the left data X and 8-bits round key K passes through S-Box.

The result is XORed with of corresponding 8-bits of right data Y. This is repeated depending on the number of XORs required to complete the P-function. XORing the output from S-box with right data saves storage required for the intermediate values. For this reason a dual port DRAM is used. The swapping of data is accomplished by addressing. It takes 44 clock cycles to complete one round. The multiplexers before the 1^{st} XOR and after the 2^{nd} XOR operation enable the computation of the modified key from the original key and the pre-and post whitening operation.

4.3.2 FL and FL $^{-1}$

The FL function breaks its 64-bit input into two 32-bit halves namely X_L and $X_R$and similarly 64-bit key kl as kl_L and kl_R. The FL ‘is broken into two parts FL_1 and FL_2 and FL^{-1} function into FL_1^{-1} and FL_2^{-1}.

$$FL_1(X_L, X_R, kl_L) = ((X_L \cap kl_L) \ggg_1) \oplus X_R \tag{4.4}$$

$$FL_2(X_L, X_R, kl_R) = (X_R \cup kl_R) \oplus X_L \tag{4.5}$$

$$FL_1^{-1}(X_L, X_R, kl_R) = (X_R \cup kl_R) \oplus X_L \tag{4.6}$$

$$FL_2^{-1}(X_L, X_R, kl_L) = ((X_L \cap kl_L) \lll_1) \oplus X_R \tag{4.7}$$

As can be seen from equations 4.4 and 4.7, FL_1 and FL_2^{-1} are the same operation and Similarly FL_2 and FL_1^{-1} from equations 4.5 and 4.6. Hence, we combine FL_1 and FL_2^{-1} as one function called FLM_1 and FL_2 and FL_1^{-1} as FLM_2.

$$FLM_1(X_L, X_R, kl_L) = ((X_L \cap kl_L) \lll_1) \oplus X_R \tag{4.8}$$

$$FLM_2(X_L, X_R, kl_R) = (X_R \cup kl_R) \oplus X_L \tag{4.9}$$

This saves two XORs needed for FL and FL^{-1} operations. The 32-bit cyclic rotation in FLM_1 is implemented as a 1-bit left shift on 8-bit data with one flipflop to store the shifted bit. After completing FLM_1, the last bit is computed again to get the correct bit.

4.3.3 Key Storage and Scheduling

The Camellia algorithm needs two keys of size 128 bit, the original key K_L and the modified key K_A, which are rotated to generate the round keys. They both are stored in 128-bit shift registers. However, such a shift register has only a single bit output and each output or tab requires a flip-flop. Hence, the area consumed by such a shift register depends mainly on the number of taps required to access the data. All the shift registers in this implementation shift by 8-bit in order to match the width of the datapath. However, the rotations needed for round key generation are 15, 30, 45, 60, 77, 94 and 111-bits. as shown in Table 4.3. We can accomplish this by 8-bit shifts and an 8-bit 5:1 multiplexer as $n \bmod 8$ has only 5 different results. In order to make the control logic simple and uniform, shifting of the key is done at the last clock cycle of the round. For normal round key generation, tapping 15-bits is sufficient. However, due to FLM_1 which has a 32-rotation, 41-bits additional tabs are required. This increased the size of the shift register approximately by 2 folds. The key scheduling can be seen in Fig 4.5. The original key K_L is initially loaded into both DRAM and K_L shift register. The constants for generating modified key K_A are stored in a separate shift register. K_A is computed using the datapath from Fig 4.4. It is loaded into the K_A shift register, while data is loaded into DRAM. Using shift registers for both K_L and K_A reduces the area by about 75% compared to using two 128-bit registers.

4.3.4 Controller

The control unit consists of a main controller and sub controllers for the F- and FLM functions. The main controller stores its control words in as Distributed ROM (DROM) for the reasons stated in Section 4.3.1. The address for the control word is generated by a 6-bit counter. Using a DROM and a counter for the main controller, its size is reduced by 97% as compared to a FSM. The sub controllers stores their control words in a shift registers as they repeat a sequence of operations.

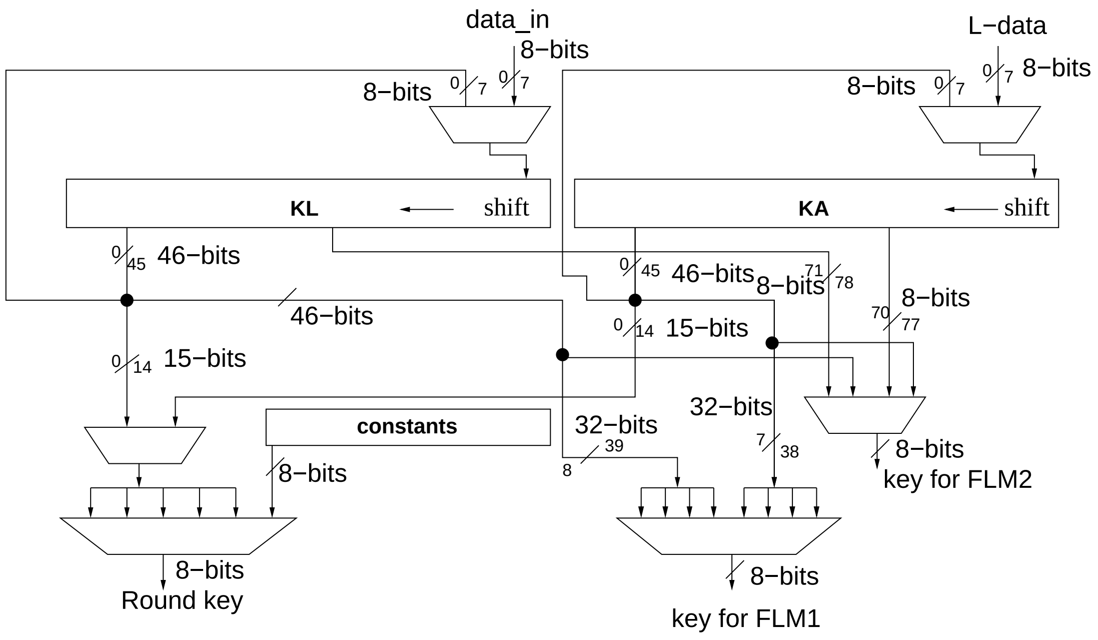

Figure 4.5: Key scheduling using shift registers

Table 4.3: Number of shifts of shift register

Round keys	Rotation of key (n)	Number of 8-bit shifts	Relative shifts	n mod 8
kw_1,kw_2,k_1,k_2	0	0	0	0
k_3,k_4,k_5,k_6	15	1	1	7
kl_1,kl_2	30	3	2	6
k_7,k_8,k_9	45	5	2	5
k_{10},k_{11},k_{12}	60	7	2	4
kl_3,kl_4	77	9	2	5
k_{13},k_{14},k_{15},k_{16}	94	11	2	6
kw_3,kw_4,k_{17},k_{18}	111	13	2	7

Table 4.4: Components for Camellia implementation

Component	Implementation	slices
F-controller-(1)	FSM	40
F-controller-(2)	Shift Register	2
Key storage-(1)	Register	128
Key Storage-(2)	Shift Register	32
Controller-(1)	FSM	214
Controller-(2)	DROM	40
Data storage-(1)	Register	64
Data storage-(2)	DRAM	16

Chapter 5: Present

5.1 Introduction

Present is an ultra-lightweight block cipher proposed by A.Bogdanov, L.R.Knuden and G. Lender et al [2]. Present is a 31-round Substitution-Permutation (SP) network with a block size of 64-bit and 80-bit or 128-bit key lengths. In this thesis a 128-bit key length is considered. Present was designed with area and power constraints as the basic design goals without a compromise in security. Present was designed by incorporating the features of Serpent [17] and Data Encryption Standard (DES) [18] which demonstrated excellent performance in hardware. The non-linear substitution layer, S-box in Present is similar to that of Serpent and the linear permutation layer to that of DES. The original Present proposal provides basic security analysis [2]. Further more analysis is performed in [19], [11] and [20] proving 31-round Present still secure.

5.2 Specification

5.2.1 Notations

P	46-bit plaintex
C	64-bit ciphertext
K	128-bit user supplied key
k_i	64-bit i_{th} round key

Present encrypts a 64-bit plaintext P to a 64-bit ciphertext C in 31-rounds of SP network and a post whitening round using 32-round keys k generated from 128-bit input key K. The

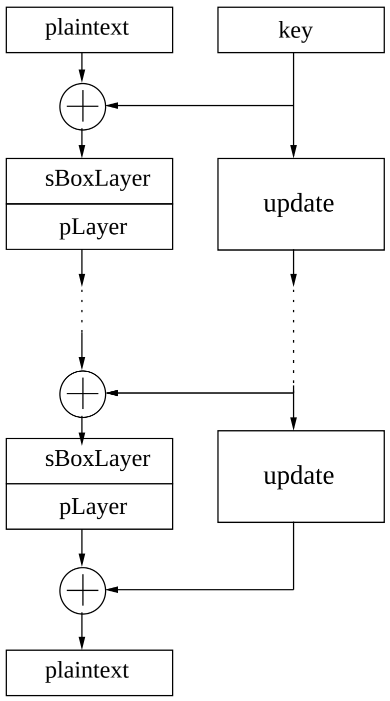

Figure 5.1: Top level algorithmic description of present

three stages addRoundkey, sBoxLayer and pLayer are involved in each round of Present. The overall Present algorithm is shown in Fig 5.1.

```
generateRoundkeys()
for $i = 1$ to 31 do
    addRoundKey(STATE,$K_i$)
    sBoxLayer(STATE)
    pLayer(STATE)
end for
addRoundKey(STATE,$K_{32}$)
```

5.2.2 addRoundKey

The addRoundkey operation introduces the round key which gets XORed with current STATE $b_{63} \cdots b_0$. For the given round key $k_i = k_{63}^i \cdots k_0^i$ for $1 \leq i \leq 32$ and the current state b_{63}......b_0 addRoundKey consists of the operation for $0 \leq j \leq 63$

Table 5.1: S-box Table

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

$$b_j \leftarrow b_j \oplus k_j^i. \tag{5.1}$$

5.2.3 sBoxLayer

The second stage is non-linear sBoxLayer which consist of 4-bit to 4-bit S-boxes which are given in hexadecimal notation in Table 5.1. During sBoxLayer operation, the 64-bit current state $b_{63} \cdots b_0$ is a concatenation of sixteen 4-bit words $w_15, \cdots, w_0$ where w_i=$b_{4*i+3}||b_{4*i+2}||b_{4*i+1}||b_{4*i}$ for $0 \leq i \leq$15 and output S(w_i) gives the updated state value.

$$For 0 \leq i \leq 15$$
$$b_{4*+3}b_{4*+2}b_{4*+1}b_{4*i} \leftarrow S(w_i). \tag{5.2}$$

5.2.4 pLayer

The linear bit permutation pLayer is third stage of the round operation. The bit permutation of Present can be described in equations 5.3, 5.4, 5.5 and 5.6.

$$For 0 \leq i \leq 15$$
$$b_i \quad \leftarrow b_{4*i} \tag{5.3}$$
$$b_{i+16} \quad \leftarrow b_{4*i+1} \tag{5.4}$$
$$b_{i+32} \quad \leftarrow b_{4*i+2} \tag{5.5}$$
$$b_{i+48} \quad \leftarrow b_{4*i+3} \tag{5.6}$$

Table 5.2: Table of pLayer

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	2

Bit permutation is also shown in Table 5.2

5.2.5 Key schedule

The user-supplied or original 128-bit key K is stored in a register and represented as $k_{127}k_{126}\cdots k_1k_0$. For the round i, the left most 64 bits of the current state of register K are the round key. So for the round i,

$$k_i = k_{63}k_{62}\cdots k_1k_0 = k_{127}k_{126}\cdots k_{65}k_{64}. \tag{5.7}$$

After the round key k_i is extracted, the key register K=$k_{127}k_{126}\cdots k_1k_0$ is shifted to the left by 61 bits and left most 8-bits are passed through two S-boxes of Present. The 5-bits $k_{66}k_{65}k_{64}k_{63}$ are XORed with 5-bit round counter.

1. $[k_{127}k_{126}\cdots k_1k_0]$=$[k_{66}k_{65}\cdots k_{68}k_{67}]$

2. $[k_{127}k_{126}k_{125}k_{124}]$=S$[k_{127}k_{126}k_{125}k_{124}]$

3. $[k_{123}k_{122}k_{121}k_{120}]$=S$[k_{123}k_{122}k_{121}k_{120}]$

4. $[k_{66}k_{65}k_{64}k_{63}]$=$[k_{66}k_{65}k_{64}k_{63}]\oplus round\_counter$

5.3 Light Weight Architecture of Present

Most the area is usually consumed for key and data storage. Reduction of area of the architecture is achieved by scaling the 64-bit implementation to a 16-bit implementation

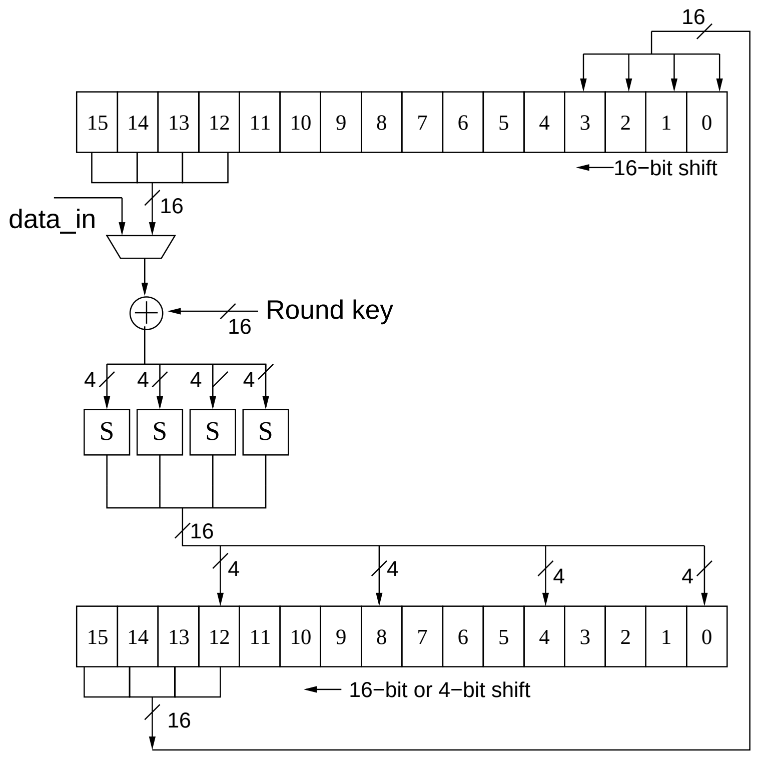

Figure 5.2: 16-bit datapath of Present

and by applying the optimization techniques stated in Chapter 2 for different components used in this implementation. Scaling to 8-bit would decrease the throughput drastically with very small reduction area due to overhead involved for performing permutation operation. The top level datapath is shown in Fig 5.2

5.3.1 Data storage

The 64-bit data is stored in a shift register for the reason specified in 2.1.1 which performs a 16-bit circular left-shift. We consider the 64-bit shift register as a combination of sixteen 4-bit block 15,···,0. To perform the round operation, the 16 most significant bits (MSB) are tapped out of shift register.

5.3.2 S-box Implementation

Present consists of 4-bit to 4-bit S-boxes which can be implemented by using a single LUT each. This architecture uses a total of six S-boxes, four for round operations and two for key scheduling.

5.3.3 Permutation Layer

When the 16 MSB bits are given to the four S-boxes and the output from S-boxes to the linear-permutation pLayer, 4-bits blocks 15, 11, 7 and 3 are computed. In next clock cycle, blocks 14, 10, 6 and 2 are computed. So in order to arrange this blocks in the correct order for the next round, another 64-bit shift register which preform a either 4-bit or 16-bit shifts is used. The input to the shift register are given to blocks 12, 8, 4 and 0. The block 15, 11, 7, and 3 are placed in block positions of 12, 8, 4 and 0. In the next clock cycle, the data in the shift register is shifted by 4-bits which moves the blocks 15, 11, 7 and 3 to positions 13, 9, 5 and 1 respectively. The blocks 14, 10, 6 and 2 computed in that clock cycle are placed in 12, 8, 4 and 0 aligning with the previous blocks 15, 11, 7 and 3. The same process repeated for another two clock cycles for computing and aligning the remaining blocks. When all the blocks of data are computed, the data is loaded back into the initial shift register during which both shift registers perform 16-bit shifts. Loading of 64-bit data back into the initial shift register would require additional clock cycles. So each round operation requires eight clock cycles.

5.3.4 Key Storage and Scheduling

The key is stored in a 128-bit shift register which performs a 16-bit circular left-shift. The initial round key are the 64-MSB bits of the original key. During first 4-clock cycles, the first round key is obtained in 4 blocks of 16-bit each and the key is shifted by 64-bits. However for next round key generation, the original key is shifted by 61 bits and 8 MSB bits of the resultant key are applied to two S-boxes. Finally, 5-bits of the round counter are XORed with bits 66,65,$\cdots$,62. The resulting 64-MSB bits are the next round key. Using 16-bit shifts, a 61-bit shift is not possible. This problem is solved by placing 3-extra taps on shift register and use of two 3-bit registers A and B along with multiplexers. After the first round of operation, the key is misaligned by 3-bits. To compensate this, register

A stores 3 LSB bits of shift register output of the previous clock cycle. Those 3-bits are concatenated as MSB bits with 16-bit shift register output resulting in total number of bits to 19. The 3-LSB bits of the 19-bits are stored in A and the remaining 16 bits are given as input to round key generator block denoted by RK as shown in Fig 5.3. This can be viewed as 3-bit shift on 19-bit key resulting in the 16-bits needed for generating the first block of the round key. This results in a total shift of 64-3=61 bit shift. The RK block consists of two S-boxes for S-box operation, 5-bit XOR to perform XOR operation with round counter and multiplexers to choose the appropriate bits for round key generation. The 13 MSB bit of the shift register, 3-bits from register A, 3-bit from register B and 5-bits of the round counter are the inputs to the RK block. The RK block performs one of the three operation, S-box operation or XOR operation with round counter or bypass both operations. The output of RK block is the round key. There is misalignment of 6-bit between any two consecutive round key generation with the exception of the first round. This offset of 6-bit is compensated by using another 3-bit register B. So using multiplexer M2, M3 and M5 along with B, the the 3-bit shift is performed on the 19-bits data (16-bits from the RK block and the 3-bits either from B or shift register output). The resulting 16 MSB bits are given as input to the shift register. With this the 3-bit offset is compensated. Another 3-bit are compensated by performing another 3-bit shift using 3-bits of A and 16-bits of shift register output.

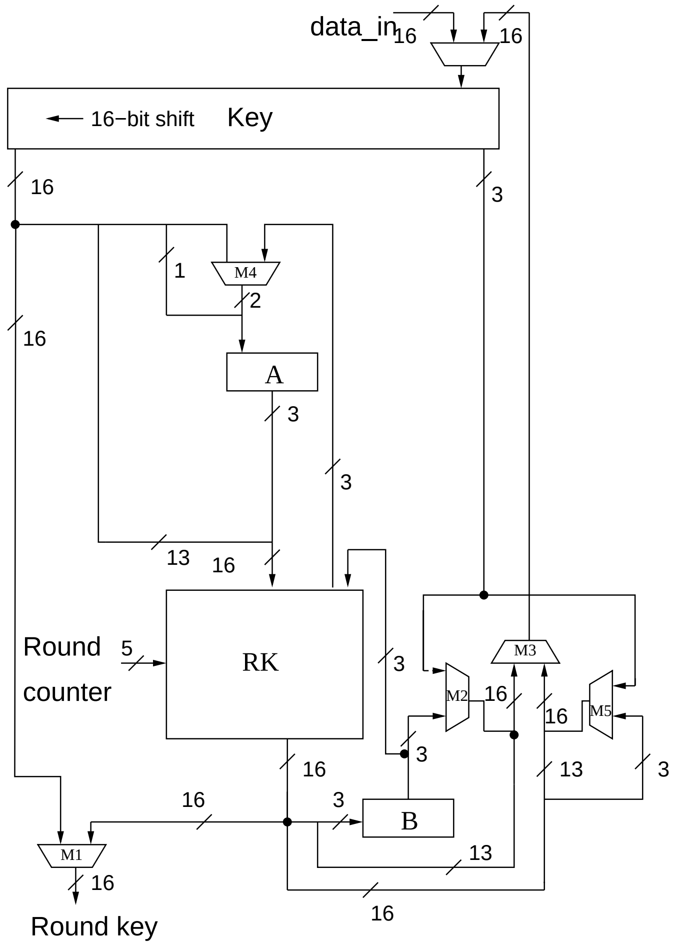

Figure 5.3: Key scheduling of Present

Chapter 6: AES and xTEA

6.1 AES

6.1.1 Introduction

AES was selected and standardized by National Institute of Standards and Technology (NIST)as a Federal Processing standard FIPS-197 [21] in 2001. AES is a 128-bit block cipher with 128-, 192- and 256-bit key length. In this implementation,a 128-bit key is considered. AES is throughly scrutinized and still considered to be secure. AES with 128-key has 10 rounds apart from initial round key addition operation. Each round of operation except the last round consists of four operation SubBytes, ShiftRows, MixColumn and AddRoundKey. In the very last round, mixcolumn operation is skipped. AES algorithm preforms operations on two-dimensional array of bytes call the State matrix.

6.1.2 Architecture and Implementation

This architecture was developed for ultra-low power applications for ASIC in [5]. In this implementation, the input data and the key are stored in memory. Each AES transformation and the key expansion load their operand in a specific order from the state memory or key memory respectively and write them back. This process is streamlined by grouping the AES transformation into four stages:

1. Initial ADDRoundKEY-SubBytes-ShiftRows
2. MixColums
3. AddRoundKey-SubBytes-ShiftRows
4. FinalAddRoundkey

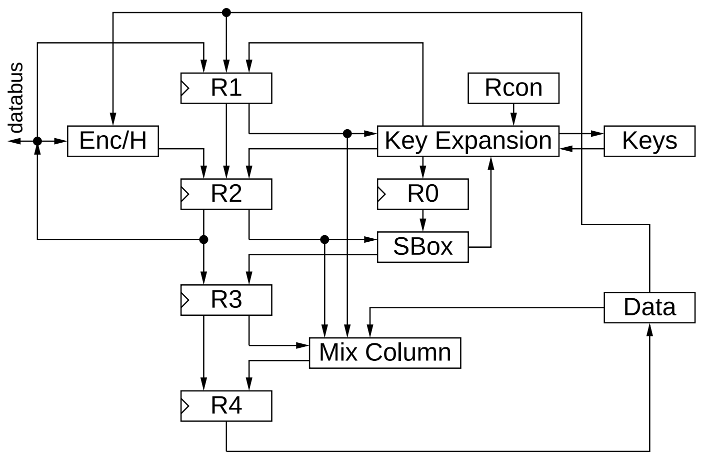

Figure 6.1: Top level block diagram of AES Datapth

This grouping allows re-use of registers and minimize the number of internal memory access. This architecture has five 8-bit register R_0, R_1, R_2, R_3 and R_4. Register R_0 is used exclusively for key storage and R_2, R_3 and R_4 for state computations. R_1 is used for key computations except during MixCloumns operation. The boxes labeled $Keys$ and $Data$ are the register files for the Round Keys and the State Memory respectively.

In the Initial ADDRoundkey-SubBytes-ShiftRows stage, the 128-bit data and the secret key are read from the memory and ADDRoundkey, SubBytes and ShiftRows operations are performed. The ADDRoundkey-SubBytes-ShiftRows is run for nine times with round keys generated on the fly for ADDRoundKey. MixColumns are implemented by using the method in [22]. In Final AddRoundkey stage, computing final round key and AddRoundkey operation are performed.

6.2 xTEA

6.2.1 Introduction

The Tiny Encryption Algorithm (TEA) was developed by David Wheeler and Roger Needham and presented in an unpublished technical report[23]. The authors eliminated the weaknesses later found in the original TEA and called eXtended TEA (xTEA) for extension of TEA [24]. xTEA uses simple addition, XOR and shifts operations and has a small size of code. These features make it ideal candidate for sensor nodes which have limited memory and computational power. TEA was developed for software implementations but its simple design makes it suitable for hardware implementations. TEA implementation of both software and hardware are reported in [25], [26], [27], [28], and [4]. TEA for sensor nodes are reported in TEA was designed for software implementation. Cryptanalysis were performed on reduced round of xTEA in [29],[30]. The 32-round version of xTEA is considered still to be secure.

6.2.2 xTEA Encryption Algorithm

Notations

$\ll x$	logical left shift by x-bits.
$\gg x$	logical right shift by x-bits.
$\oplus$	bitwise XOR.

Algorithm and Implementation Architecture

xTEA is a 64-bit block cipher with 128-bit key length. The number of rounds typically are 32. Each of the 64-bit input blocks are split into two halves y and z. These two halves are applied to the Feistel network and then mixed using integer addition modulo 2^{32}. The variable z, y, and sum are of 32-bit size. The formula used for computing the new values of y and Z is the same for both consisting permutation function and a subkey generation are shown in 6.1 and 6.2.

G eneration of subkey consists the function k(sum) which selects the one of the four 32-block of the original key depending on either bits 1 and 0, or bits 12 and 11 of the variable sum. The subkey sk is XORed with output of the permutation function to generate the new value. The simplified block diagram of the xTEA is shown in Fig 6.2. One round of TEA computes new values of y and Z . Computation of each value can be viewed as half-round as the same functions are used for both. A new value of sum is computed between first and

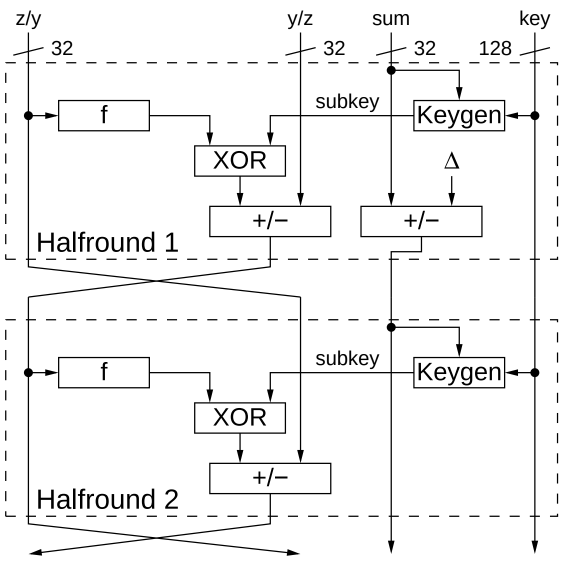

Figure 6.2: Top level block diagram of xTEA

second half-round. It is incremented by a constant δ. This architecture was developed and implemented on both ASICS and FPGAs in [4].

$$(z) = (z \ll 4 \oplus z \gg 5) + z \tag{6.1}$$

$$f(y) = (y \ll 4 \oplus y \gg 5) + y \tag{6.2}$$

$$sk = sum + k(sum) \tag{6.3}$$

Chapter 7: Implementation Results

7.1 Tools Used

All the implementation described in Chapters 3, 4, and 5 are implemented on the smallest device of the Xilinx Spartan-3 FPGA-XC3S50-5 using Xilinx ISE 9.2i for synthesis and Active HDL 7.2 for simulation.

7.2 Comparison of Block cipher Elements

The Table 7.3 gives an overview of the functional elements of the block ciphers considered in this thesis. AES has the least number of rounds while HIGHT and xTEA have the highest number of rounds. AES, Camellia, and HIGHT use pre-whitening rounds while xTEA and Present do not. Camellia, Present, and HIGHT have post-whitening rounds while AES and xTEA do not. Of all the five ciphers , xTEA does have neither pre- nor post-whitening rounds. xTEA and Camellia have a traditional feistel structure while HIGHT uses a different feistel structure. AES, Camellia, and Present have SP networks for mixing the bits while xTEA and HIGHT have addition modular 2W, XOR, and bitwise wise rotation operations for mixing. AES uses eight 8x8 Sboxes of same type while Camellia uses eight 8x8 Sboxes chosen from four different types. Present uses sixteen 4x4 Sboxes.

7.3 Results and Analysis of Light Weight Implementations

Implementing an 8x8 Sbox needs 64 slices while 4x4 Sbox needs 2 slices. This shows use of 4x4 Sbox makes the design small. AES and Camellia occupies more area even though they are scaled down to 8-bit implementations as they both use 8x8 Sboxes. In these implementations AES and Camellia use only one Sbox. The Sboxes of AES and Camellia occupy 16% and 20% of total design area. Our implementation of Present uses six 4x4 Sboxexs which occupy 10% of total design area. The AES and xTEA implementations use register for data and key storage which leads to high area consumption. Camellia uses a dual-port DRAM for data and two shift register for storing its two keys. The total area of Camellia was huge due to 15- or 17-bit shifts in round key generatation. Implementing shift which are multiples of 8 require less area. The same can be observed in Present's key schedule as its involves 61-bit shifts. Implementing permutation functions also increase the area consumption and latency for light weight implemenations in FPGA. The latency of AES and Camellia increased due to the permutation function resulting in reduced throughput. Increase in area consumption by 25% and latency by nearly 93% are observed in Present due to the permutation function. All results of light weight implemenations are summarized in Table 7.2. TinyxTEA-3[5] has the highest throughput while AES [4] has the lowest. Even though delay of Camellia is low, its throughput is reduced due to its high latency. Present outperformed in throughput/area with 0.24Mbps/slice followed by HIGHT with 0.21Mbps as these two are developed specifically for light weight cryptography.

In Table 7.3 we also compare our implementations with other smallest implementation of block cipher along with the Estream portfolio stream ciphers. Table 7.3 consists of four cluster, implementation developed in this thesis, implementations of AES and TinyxTEA by one group, all block cipher implementations and finally stream ciphers. We believe our implementation of Present and HIGHT only FPGA implementations to date. The stream cipher have very higher throughput/area than all block cipher implementations while maintaining a small area. Our implementations are so tiny that they can be used for light weight cryptographic applications.

Table 7.1: Block cipher elements

Cipher	Block size	Key size	Number of rounds	Feistel	SP network	Addition mod 2w	S-boxes
AES	128	128	10		Yes		Yes
xTEA	64	128	32	Yes		Yes	
Camellia	128	128	18	Yes	Yes		Yes
Present	64	128	31		Yes		Yes
HIGHT	64	128	32	Yes		Yes	

Table 7.2: Light weight implementation results

Cipher	Flipflops	LUTs	RAMs	Shift registers	DATA	Key	Area(slices)	Delay(ns)	Clock cycles	Throughput (Mbps)	Throughput/ Area(Mbps/slice)
AES[5]	338	531	0	0	Register	Register	393	14.21	534	16.86	0.04
TinyxTEA-3[4]	226	424	0	0	Register	Register	254	15.97	112	35.79	0.14
TinyxTEA-1[4]	259	513	0	0	Register	Register	266	13.87	240	19.23	0.07
Camellia	164	420	16	88	DRAM	Shift register	318	7.95	875	18.41	0.06
Present	114	159	0	35	Shift register	Shift register	117	8.783	256	28.46	0.24
HIGHT	34	149	8	53	Shift register	DRAM	97	11.17	279	20.53	0.21

Table 7.3: Results of our light weight implementation compared to other smallest implementation of block ciphers and the eSTREAM Portfolio ciphers on FPGA

Design	Maximum Delay (ns)	Clock Cycles	Block Size (bits)	Key Size (bits)	Area (slices)	Block RAMs	Throughput (Mbps)	Throughput/ Area (Mbps/slice)	Device
Camellia	7.95	875	128	128	318	0	18.41	0.06	xc3s50-5
Camellia	11.01	875	128	128	399	0	13.28	0.03	xc2s30-6
Present	8.783	256	64	128	117	0	28.46	0.24	xc3s50-5
HIGHT	11.17	279	64	128	97	0	20.53	0.21	xc3s50-5
AES[5]	14.21	534	128	128	393	0	16.86	0.04	Xc3s50-5
TinyXTEA-1 [4]	13.87	240	64	128	266	0	19.22	0.07	xc3s50-5
TinyXTEA-3 [4]	15.97	112	64	128	254	0	35.78	0.14	xc3s50-5
Camellia [12]	22.62	44	128	128	908	0	128.58	0.14	Xilinx VirtexE
Camellia [31]	18.28	18	128	128	1023	8	388.9	0.25	xc3s1000
Camellia [31]	17.34	18	128	128	1031	8	410.5	0.27	xc3s1000
AES 8-bit[32]	14.93	3900	128	128	124	2	2.2	0.01	xc2s15-6
AES [33]	16.67	46	128	128	222	3	166	0.32	xc2s30-6
Grain v1 [34]	5.10	1	1	80	44	0	196	4.45	xc3s50-5
Grain 128 [34]	5.10	1	1	128	50	0	196	3.92	xc3s50-5
MICKEY v2 [34]	4.29	1	1	80	115	0	233	2.03	xc3s50-5
MICKEY 128 [34]	4.48	1	1	128	176	0	223	1.27	xc3s50-5
Trivium [34]	4.17	1	1	80	50	0	240	4.80	xc3s50-5
Trivium (x64) [34]	4.74	1	64	80	344	0	13,504	39.26	xc3s400-5

Chapter 8: Differential Power Analysis Attacks

8.1 Test Equipment

In order to perform the DPA attack, the light weight architectures are implemented on Xilinx Spartan3e starter board containing a XC3S500e-FG320-4 FPGA. The core voltage net capacitances are removed for better data dependency power traces. The external power supply unit is used for core voltages of FPGA. Power consumptions CMOSare measured using a Tektronics CT-1 current probe and an Agilent DSO6054A oscilloscope, which has a bandwidth of 500MHz and sampling rate of 4GSa/sec.

8.2 Attack Setup

A wrapper is build on top of the algorithm for performing the DPA attack. Wrapper consist of 2x1 multiplexer for selecting key and data inputs to the algorithm along with a LFSR and two counters. Random Plaintexts are generated by using an LFSR either 64-bit or 128-bit depending on the block size needed. 2000 plaintexts are generated using the same seed for LFSR. A different seed is employed for next set of plaintexts. One of the two counter is used for generating trigger signal to identify the appropriate clock cycle for attack. We use hamming distance model for all the attacks described in chapter 1. The algorithm are not executed until the very end but are always reset after few cycles of trigger signal generation and new data is loaded.

8.2.1 Notations Used

$$
\begin{aligned}
K &= \text{Actual key} \\
K_{guess} &= \text{key guess } 0 \leq k_{guess} \leq 0\text{xff} \\
P &= \text{input plaintext} \\
I &= \text{intermediate result} \\
P_m &= \text{Measured power} \\
P_s &= \text{ Hypothetical power model values}
\end{aligned}
$$

8.3 DPA Attack on Camellia

Architecture of Camellia presented in this thesis makes it difficult to attack on pre-whitening round. Modified key K_A is computed using the datapath of round function and stored initially in DRAM. K_A is loaded into the shift register while the data XORed with pre-whitening key is written into that location. The data in DRAM is unknown and the incoming known data gets XORed with unknown pre-whitening key. The new data and the previous data in that location is unknown which makes the attack difficult. We choose to attack the round function to access the strength of round function against DPA, assuming there is no pre-whitening round. With no pre-whitening round, known data is loaded into DRAM. During the first clock cycle of the round operation, 8-bits of known data denoted by P_1 is XORed with 8-bit of unknown round key K. The result is applied to S-box data then XORed with another 8-bits of known data denoted by P_2. The result is written into address location of P_2. We attack at this point when the data is being written into that location. This is repeated for all the 2000 encryptions performed. The hypothetical intermediate values are calculated using K_{guess} as the key for different plain text inputs. The values of K_{guess} are varied from 0 to 255. P_s is calculated using the hamming distance model using equations 8.1 and 8.2. The P_m and P_s are correlated using equation 1.2 and the plot can be seen in Fig 8.1. The attack was successful with 2000 power traces. This attack can be repeated for calculation other key bits.

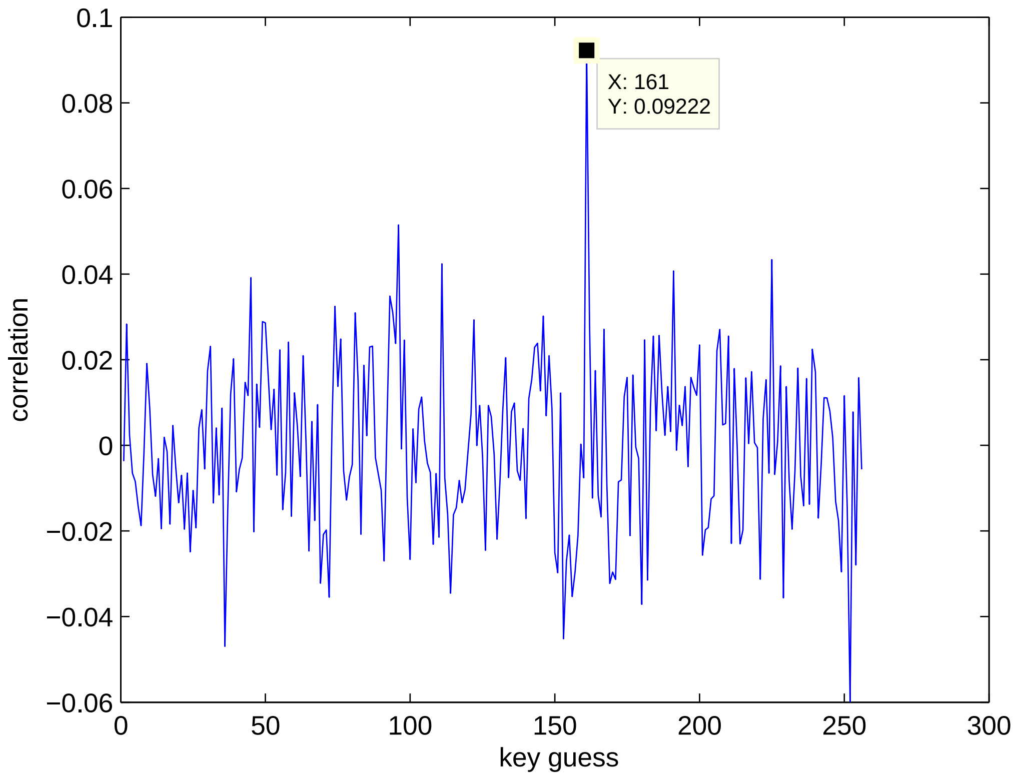

Figure 8.1: Correlation plot for attack on round function of camellia

$$I = (sbox(P_1 \oplus k_{guess})) \oplus P_2) \tag{8.1}$$

$$P_s = HD(I, P_2) \tag{8.2}$$

8.4 DPA Attack on HIGHT

The attack on HIGHT is performed on initial transformation round. We performed attack two attacks one on an addition and the other on XOR operation of initial transformation . Let P_1, P_2, P_3 and P_4 be the consecutive 8-bit data inputs. Let I_a be the intermediate result for addition and I_x for XOR operation. Initially the shift register to store data is set to zeros. In the first case, we attack when P_2 is XORed with whitening key and the result is written into block location 6 shown in Fig 3.4. The change of data in block 6 is from P_1 to I_x during this operation. For the second case, we attack when P_4 is added to whitening key and the result is written into block 6. During this operation, the data changes from P_3 to P_4. The P_{sx} and P_{sa} represents hypothetical values for XOR and addition operation respectively. Values of P_{sx} and P_{sa} are calculated using equations 8.3, 8.4, 8.5 and 8.6. The correlation plot for XOR operation attack have patterns but has a peak at the correct key guess. The correlation plot for addition operation also involves patterns which did not have any peak at the correct key guess. The correlation plots for 2000 power traces of XOR and addition operation are shown in Fig 8.2 and Fig 8.3 respectively.

$$I_x = P_4 \boxplus K_{guess} \tag{8.3}$$

$$P_{sx} = HD(I_x, P_1) \tag{8.4}$$

$$I_a = P_2 \oplus K_{guess} \tag{8.5}$$

$$P_{sa} = HD(I_a, P_3) \tag{8.6}$$

8.5 DPA Attack on xTEA

We perform the attack after the first half round operation. The attack point is when result of the half round is being written into Z. The plaintext input P here is a 32-bit input as xTEA is a 32-bit implementation. Let the T denotes the f(z) given by 6.1. For calculation of P_s we only consider 8 MSB bits of the T denoted by T_1. Let 8 MSB bits of P be denoted by P_1 The calculation of P_s are given by equations 8.7, 8.8 and 8.9. The correlation plot for 1000 power traces is given by Fig 8.4.

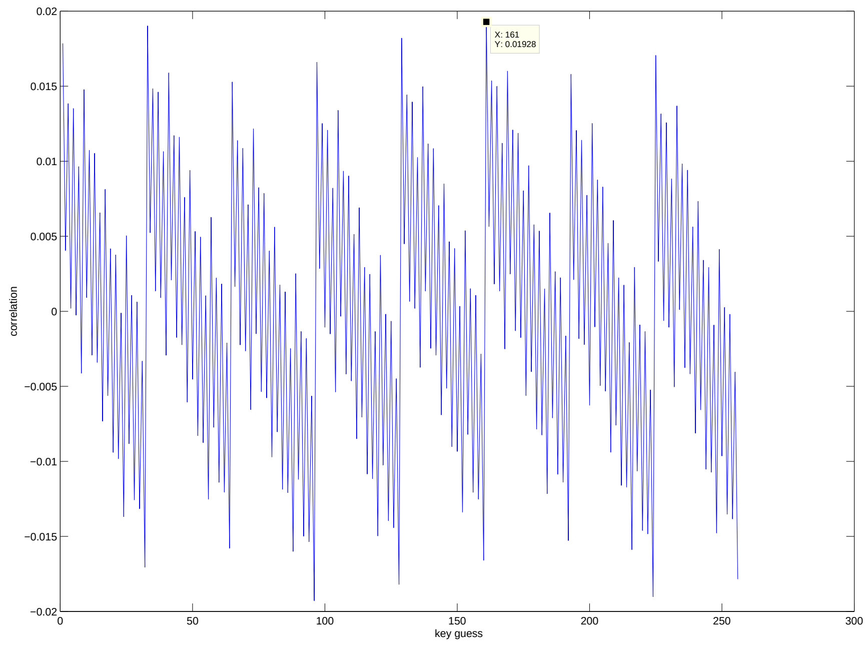

Figure 8.2: Correlation plot for attack on XOR operation in HIGHT

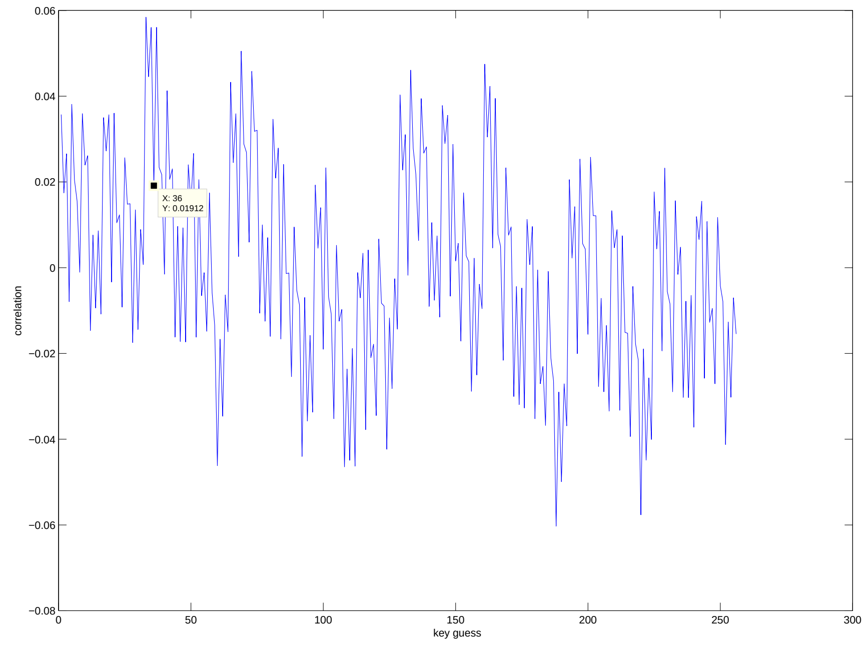

Figure 8.3: Correlation plot for addition operation in initial transformation round of HIGHT

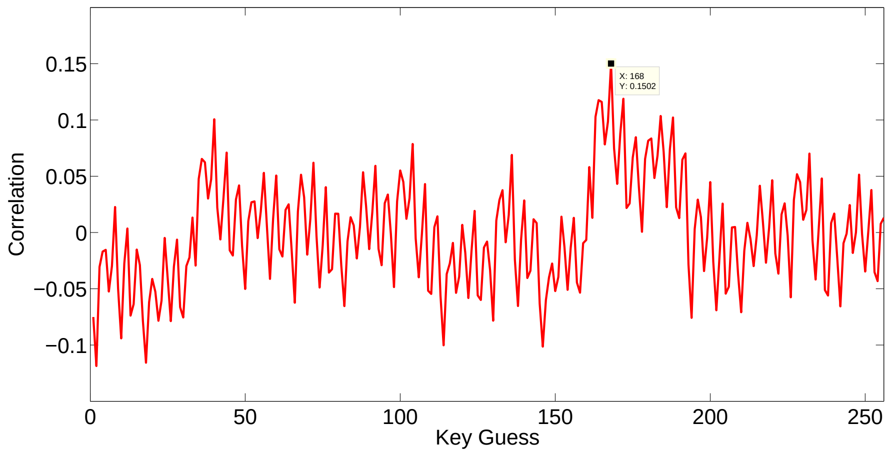

Figure 8.4: Correlation plot of xTEA

$$T = ((P \ll 4 \oplus P \gg 5) + P) \tag{8.7}$$

$$I = T_1 \oplus K_{guess} \tag{8.8}$$

$$P_s = HD(I, P_1) \tag{8.9}$$

8.6 DPA Attack on AES

The attack on AES is performed on computation of 8-bits of first round operation. On reset the data in register R_2, R_3 and R_4 in Fig 6.1 are set 0x00. In the next clock cycle, the output of register R_3 changes from 0x00 to 0x63 as its input is Sbox(R_2). The output of R_2 is 0x00. The value in R_2 changes from $0x00$ to input data P. In the next clock cycle R_3 changes from $0x00$ to Sbox($P \oplus K$). This is the clock cycle the attack is performed on R_3. P_s values are calculated using equations 8.10 and 8.11. The correlation plot for this attack is shown in Fig 8.5.

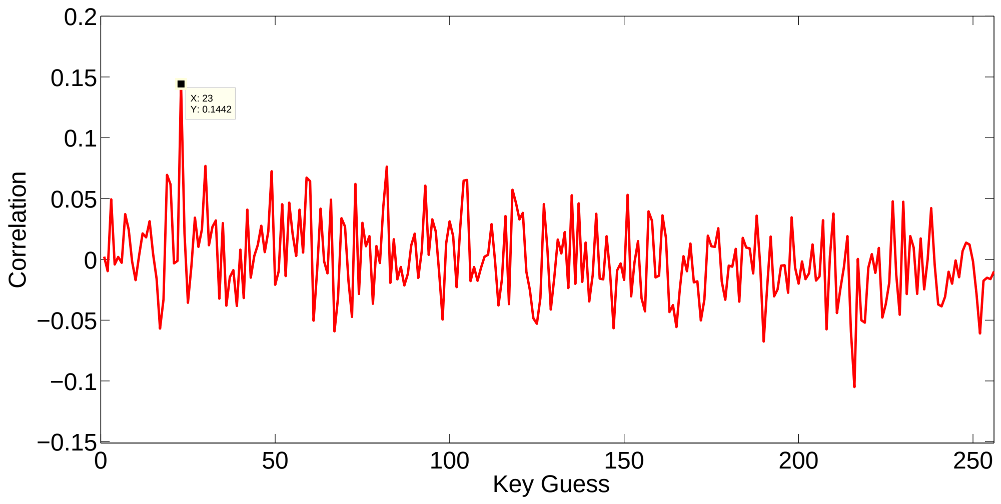

Figure 8.5: Correlation plot of AES

$$I = Sbox(P \oplus K_{guess}) \tag{8.10}$$

$$P_s = HD(I, 0x63) \tag{8.11}$$

8.7 DPA Attack on Present

We performed attack on first round of Present. Here the plaintext P is 16-bit data input with 8 MSB bits are considered concatenation of two 4-bit P_1, P_2. The 8-bit key guess K_{guess} is also considered concatenation of two 4-bit K_{guess1}, K_{guess2}. Initially zeros are loaded into the second shift register in Fig 5.2. So in the first clock cycle of round operation, the 16-bit plaintext is XORed with 16-bit round key and applied to four Sboxes. The resultant data is written into the second shift register. So the data in the shift register changes from all zeros to 16-bit output of first round. The P_s is calculated using equations 8.12 and 8.13. The correlation plot is given by Fig 8.6. The attack on Present was not successful as the correlation plot did not contain the peak at the correct key guess even with 3000 power traces. We believe this is due to use of shift register which have very little power

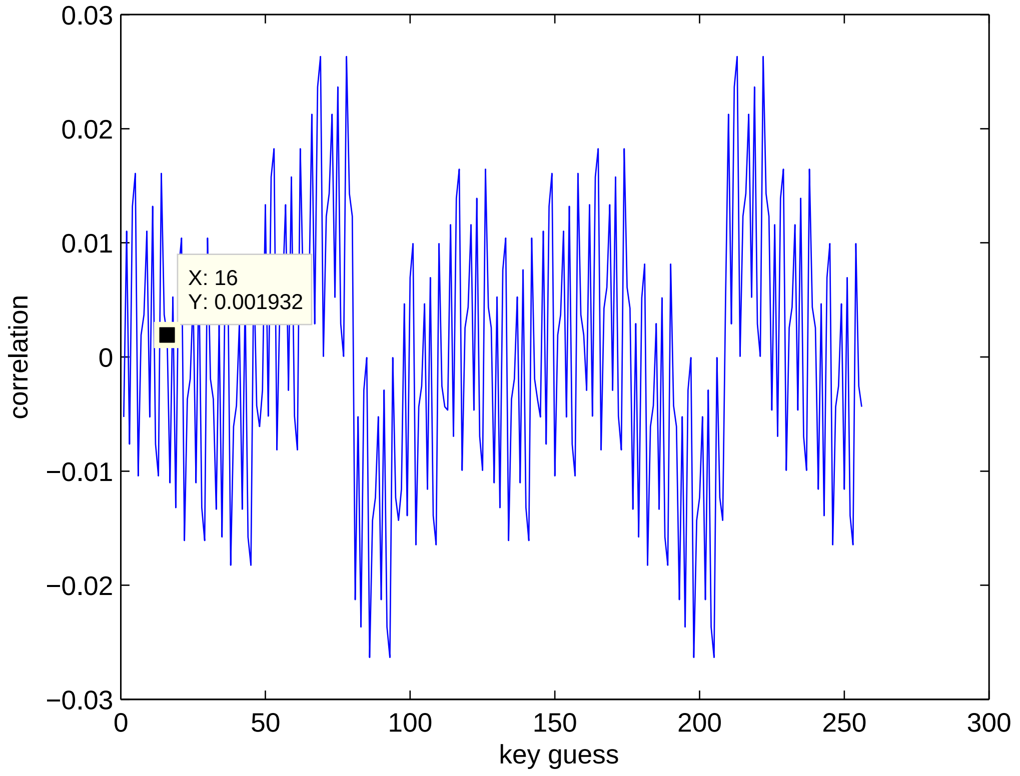

Figure 8.6: Correlation plot of Present

consumption due to change of bits.

$$I = Sbox(P_1 \oplus K_{guess1}) \parallel Sbox(P_2 \oplus K_{guess2}) \tag{8.12}$$

$$P_s = HD(0x00, I) \tag{8.13}$$

8.8 Results and Conclusions

The Table 8.1 summarizes the DPA attacks on all the five block cipher implementations. From the results , we came to the following conclusions

- Use of registers for data and key storage are more vulnerable to DPA attacks followed

Table 8.1: Result of DPA attacks

cipher	Targeted round	Number of Samples	Key revealed
Camellia	First round	3000	Yes
HIGHT	XOR-Initial Transformation	3000	Yes
HIGHT	addition-Initial Transformation	3000	No
xTEA	First half round	1000	Yes
AES	First round	1000	Yes
Present	First round	3000	No

by DRAMs and shift registers.

- Use of XOR operations make the algorithm more vulnerable to DPA than addition operations.

The natural resistance of a physical implementation of cryptographic algorithm to DPA depends on operations used in the algorithm and its implementation architecture. For example, use of shift register make them more resistant to DPA then register for storage of data and key.

Chapter 9: Future Work

In this thesis only Camellia, HIGHT and Present are optimized. These optimizations can be extended for AES and xTEA. Implementing other light weight cipher like DES, DESXL, Clefia and mCryton and analyzing them would be good area to explore. Analyzing individual components of cipher against DPA would be another another extension of this work.

Bibliography

[1] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee, "HIGHT: A new block cipher suitable for low-resource device," in *CHES 2006*, ser. LNCS, L. Goubin and M. Matsui, Eds., vol. 4249. International Association for Cryptologic Research, 2006, pp. 46–59.

[2] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems–CHES 2007*, ser. Lecture Notes in Computer Science (LNCS), vol. 4727. Springer, 2007, pp. 450–466.

[3] K. Paul, J. Joshua, and J. Benjamin, "Introduction to differential power analysis and related attacks," http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf, 1998.

[4] J.-P. Kaps, "Chai-tea, cryptographic hardware implementations of xTEA," in *INDOCRYPT 2008*, ser. LNCS, D. Chowdhury, V. Rijmen, and A. Das, Eds., vol. 5365. Heidelberg: Springer, Dec 2008, pp. 363–375.

[5] J.-P. Kaps and B. Sunar, "Energy comparison of AES and SHA-1 for ubiquitous computing," in *Embedded and Ubiquitous Computing (EUC-06) Workshop Proceedings*, ser. Lecture Notes in Computer Science (LNCS), X. Z. et al., Ed., vol. 4097. Springer, Aug 2006, pp. 372–381.

[6] M. Rawski, H. Selvaraj, and T. Luba, "An application of functional decomposition in ROM-based FSM implementation in fpga devices," *J. Syst. Archit.*, vol. 51, no. 6-7, pp. 424–434, 2005.

[7] V. Skylarov, "Synthesis and implementation of RAM-based finite state machines in FPGAs," in *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications.* London, UK: Springer-Verlag, 2000, pp. 718–728.

[8] I. Garcia-Vargas, R. Senhadji-Navarro, G. Jiménez-Moreno, A. Civit-Balcells, and P. Guerra-Gutierrez, "Rom-based finite state machine implementation in low cost FPGAs," in *Industrial Electronics, 2007 ISIE 2007. IEEE International Symposium on*, June 2007, pp. 2342–2347.

[9] A. Tiwari and K. A. Tomko, "Saving power by mapping finite-state machines into embedded memory blocks in fpgas," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe.* IEEE Computer Society, 2004, p. 20916.

[10] L. Young-Il, L. Je-Hoon, Y. Younggap, and C. Kyoung-Rok, "Implementation of HIGHT cryptic circuit for RFID tag," *IEICE Electronics Express*, vol. 6, no. 4, pp. 180–186, 2009.

[11] O. Ozen, K. Varici, C. Tezcan, and C. Kocair, "Lightweight block cipher revisisted:cryptanalysis of reduced round PRESENT and HIGHT," in *Australasian Conference on Information Security and Privacy -ACISP*, vol. 5594, 2009, pp. 90–107.

[12] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms – design and analysis," in *SAC 2000*, ser. LNCS, vol. 2012. Springer, 2001, pp. 39–56.

[13] *Final report of NESSIE*, April 2004, https://www.cosic.esat.kuleuven.be/nessie/Bookv015.pdf.

[14] J. Lu, J. Kim, N. Keller, and O. Dunkelman, "Improving the efficiency of impossible differential cryptanalysis of reduced Camellia and MISTY1," in *CT-RSA 2008*, ser. LNCS, T. Malkin, Ed., vol. 4964. Berlin: Springer-Verlag, April 2008, pp. 370–386.

[15] D. Lei, L. Chao, and K. Feng, "New observation on Camellia," in *ACM Symposium on Applied Computing 2006*, ser. LNCS, B.Preneel and S. Tavares, Eds., vol. 3897. Berlin: Springer-Verlag, February 2006, pp. 51–64.

[16] G. Jie and Z. Zhongya, "Improved collision attack on reduced round Camellia," in *CANS 2006*, ser. LNCS, D.Pointcheval, Y. Mu, and K.Chen, Eds., vol. 4301. Berlin: Springer-Verlag, November 2006, pp. 189–190.

[17] E. Biham, R. Anderson, and L. Knudsen, "Serpent: A new block cipher proposal," in *Fast Software Encryption,FSE 1998,*, ser. Lecture Notes in Computer Science (LNCS), vol. 1372. Springer,, January 1998, pp. 222–223.

[18] *Data Encryption Standard*, National Institute of Standards and Technology, FIPS Publication 46-3, October 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[19] w. Meiqin, "Differential cryptanalysis of reduced-round PRESENT," in *Cryptographic hardware and embedded systems – CHES 2008*, ser. Lecture Notes in Computer Science (LNCS), S. Vaudenay, Ed., vol. 5023. Springer, 2008, pp. 40–49.

[20] B. Collard and F.-x. Standaert, "A statistical saturation attack against the block cipher PRESENT," in *CT-RSA*, vol. 5473. Springer, 2009, pp. 195–210.

[21] *AES*, NIST, FIPS Publication 197, Nov 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[22] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, "Strong authentication for RFID systems using the AES algorithm," in *Cryptographic hardware and embedded systems – CHES 2004*, ser. Lecture Notes in Computer Science (LNCS), M. Joye and J.-J. Quisquater, Eds., vol. 3156. Springer, Aug 2004, pp. 357–370.

[23] D. Wheeler and R. Needham, "TEA, a tiny encryption algorithm," Cambridge University, England, Tech. Rep., Nov 1994.

[24] ——, "Correction to xtea," Cambridge University, Tech. Rep., 1998.

[25] S. Liu, O. V. Gavrylyako, and P. G. Bradford, "Implementing the TEA algorithm on sensors," in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference.* New York, NY, USA: ACM Press, 2004, pp. 64–69.

[26] M. Pavlin, "Encryption using low cost microcontrollers," in *42nd International Conference on Microelectronics, Devices and Materials and the Workshop on MEMS and NEMS.* Society for Microelectronics Electronic, 2006, pp. 189–194.

[27] P. Israsena, "Securing ubiquitous and low-cost RFID using tiny encryption algorithm," in *Symp. on Wireless Pervasive Computing.* IEEE, Jan 2006, 4 pp.

[28] ——, "Design and implementation of low power hardware encryption for low cost secure RFID using TEA," in *Information, Communications and Signal Processing*, Dec 2005, pp. 1402–1406.

[29] D. Moon, K. Hwang, W. Lee, S. Lee, and J. Lim, "Impossible differential cryptanalysis of reduced round XTEA and TEA," in *Fast Software Encryption, FSE 2002*, ser. LNCS, J. Daemen and V. Rijmen, Eds., vol. 2365. Springer, 2002, pp. 49–60.

[30] J. C. H. Castro and P. I. Viñuela, "New results on the genetic cryptanalysis of TEA and reduced-round versions of XTEA," in *Congress on Evolutionary Computation CEC2004*, vol. 2, 2004, pp. 2124–2129.

[31] D. Denning, I. James, and D. Malachy, "Compact iterative FPGA camellia algorithm implementation," in *FPT 2004*, December 2004, pp. 311–314.

[32] T. Good and M. Benaissa, "AES on FPGA from the fastest to the smallest." in *CHES 2005*, ser. LNCS, J. R. Rao and B. Sunar, Eds., vol. 3659. Springer, 2005, pp. 427–440.

[33] P. Chodowiec and K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *CHES 2003*, ser. LNCS, vol. 2779. Springer, Sep. 2003, pp. 319–333.

[34] D. Hwang, M. Chaney, S. Karanam, N. Ton, and K. Gaj, "Comparison of FPGA-targeted hardware implementations of eSTREAM stream cipher candidates," in *SASC 2008*, Feb 2008, pp. 151–162.

Curriculum Vitae

Panasayya Yalla received his Bachelor of Engineering degree from Sir C.R.Reddy College of Engineering, Andhra Pradesh, India in 2006. He started workign towards his master's degree in George Mason University from 2007. He was involved in teaching various undergraduate courses. He is also a member of Cryptographic Engineering Research Group (CERG).